# Exercise2 - Cameras and Lenses

## Introduction

The purpose of this exercise is to use Python to calculate camera and scene specific values.

## Learning Objectives

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

- Create a Python function that uses the thin lens equation to compute either the focal length (f), where the rays are focused (b) or an object distance (g) when two of the other measurements are given

## Computing camera parameters

### Exercise 1

Explain how to calculate the angle $\theta$ when $a$ and $b$ is given in the figure below. Calculate $\theta$ (in degrees) when $a = 10$ and $b = 3$ using the function `math.atan2()`. Remember to import `math` and find out what `atan2` does.

Angle in triangel

### Exercise 2

Create a Python function called `camera_b_distance`.

The function should accept two arguments, a focal length f and an object distance g. It should return the distance from the lens to where the rays are focused (b) (where the CCD should be placed)

The function should start like this:

```python
def camera_b_distance(f, g):
    """
    camera_b_distance returns the distance (b) where the CCD should be placed
    when the object distance (g) and the focal length (f) are given
    :param f: Focal length
    :param g: Object distance
    :return: b, the distance where the CCD should be placed
    """
```

It should be based on Gauss' lens equation:

$$\frac{1}{g} + \frac{1}{b} = \frac{1}{f}$$

You should decide if your function should calculate distances in mm or in meters, but remember to be consistent!

Use your function to find out where the CCD should be placed when the focal length is 15 mm and the object distance is 0.1, 1, 5, and 15 meters.

What happens to the place of the CCD when the object distance is increased?

## Camera exercise

In the following exercise, you should remember to explain when something is in mm and when it is in meters. To convert between radians and degrees you can use:

```
angle_degrees = 180.0 / math.pi * angle_radians
```

### Exercise 3

Thomas is 1.8 meters tall and standing 5 meters from a camera. The cameras focal length is 5 mm. The CCD in the camera can be seen in the figure below. It is a 1/2" (inches) CCD chip and the image formed by the CCD is 640x480 pixels in a (x,y) coordinate system.

CCD chip

It is easiest to start by drawing the scene. The scene should contain the optical axis, the optical center, the lens, the focal point, the CCD chip, and Thomas. Do it on paper or even better in a drawing program.

1. A focused image of Thomas is formed inside the camera. At which distance from the lens?
2. How tall (in mm) will Thomas be on the CCD-chip?
3. What is the size of a single pixel on the CCD chip? (in mm)?
4. How tall (in pixels) will Thomas be on the CCD-chip?
5. What is the horizontal field-of-view (in degrees)?
6. What is the vertical field-of-view (in degrees)?

# Exercise2b - Change detection in videos

The goal of this exercise is to create a small program for real-time change detection using OpenCV.

## Learning Objectives

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

1. Use OpenCV to access a web-camera or the camera or a mobile phone.
2. Use the OpenCV function `cvtColor` to convert from color to gray scale,
3. Convert images from integer to floating point using the `img_as_float` function.
4. Convert image from floating point to uint8 using the `img_as_ubyte` function.
5. Compute a floating point absolute difference image between a new and a previous image.
6. Compute the frames-per-second of an image analysis system.
7. Show text on an image using the OpenCV function `putText`.
8. Display an image and zoom on pixel values using the OpenCV function `imshow`.
9. Implement and test a change detection program.
10. Update a background image using a linear combination of the previous background image and a new frame.
11. Compute a binary image by thresholding an absolute difference image.
12. Compute the total number of changed pixels in a binary image.
13. Implement a simple decision algorithm that is based on counting the amount of changed pixels in an image.

## Installing Python packages

In this exercise, we will be using the popular *OpenCV* library to perform real-time image analysis.

We will use the virtual environment from the previous exercise (`course02502`). Start an **Anaconda prompt** and do:

```
activate course02502
conda install -c conda-forge opencv
```

You might also need to install **Numpy**:

```
conda install -c anaconda numpy
```

## Exercise data and material

The data and material needed for this exercise can be found here: (https://github.com/RasmusRPaulsen/DTUImageAnalysis/blob/main/exercises/ex2b-ChangeDetectionInVideos/data/)

Start by creating an exercise folder where you keep your data, Python scripts or Notebooks. Download the data and material and place them in this folder.

## OpenCV program for image differencing

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 and after that to floating point images
3. Computes a difference image between a current image and the previous image.
4. Computes the frames per second (fps) and shows it on an image.
5. Shows 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 Using a mobile phone.

Note that we sometimes refers to an image as a *frame*.

**Exercise 1:** *Run the program from the exercise material and see if shows the expected results? Try to move your hands in front of the camera and try to move the camera and see the effects on the difference image.*

**Exercise 2:** *Identify the important steps above in the program. What function is used to convert a color image to a gray-scale image?*

## Change detection by background subtraction

The goal of this exercise, is to modify the program in the exercise material, so it will be able to raise an alarm if significant changes are detected in a video stream.

The overall structure of the program should be:

- Connect to camera
- Acquire a background image, convert it to grayscale and then to floating point
- Start a loop:
    1. Acquire a new image, convert it to grayscale and then to floating point: $I_{\text{new}}$ .

2. Computes an absolute difference image between the new image and the background image.
3. Creates a binary image by applying a threshold, T, to the difference image.
4. Computes the total number of foreground, F, pixels in the foreground image.
5. Compute the percentage of foreground pixels compared to the total number of pixels in the image (F).
6. Decides if an alarm should be raised if F is larger than an alert value, A.
7. If an alarm is raised, show a text on the input image. For example **Change Detected!**.
8. Shows the input image, the backround image, the difference image, and the binary image. The binary image should be converted to uint8 using `img_as_ubyte`.
9. Updates the background image, $I_{\text{background}}$, using:

$$I_{\text{background}} = \alpha * I_{\text{background}} + (1 - \alpha) * I_{\text{new}}$$

10. Stop the loop if the key `q` is pressed.

You can start by trying with $\alpha = 0.95$, $T = 0.1$, and $A = 0.05$.

**Exercise 3:** *Implement and test the above program.*

**Exercise 4:** *Try to change* $\alpha$, $T$ and $A$. *What effects do it have?*

The images are displayed using the OpenCV function `imshow`. The display window has several ways of zooming in the displayed image. One function is to zoom x30 that shows the pixel values as numbers.

**Exercise 5:** *Try to play around with the zoom window.*

**Exercise 6:** *Use `putText` to write some important information on the image. For example the number of changed pixel, the average, minumum and maximum value in the difference image. These values can then be used to find even better values for $\alpha$, $T$ and $A$.*

Also try to find out how to put a colored text on a color image. Here you need to know that OpenCV stores color as BGR instead of RGB.

## Using a mobile phone camera (optional)

It is possible to use a mobile phone as a remote camera in OpenCV.

You need to install a web cam app on your phone. One option is `DroidCam` that can be installed from Google Play or from Apple App Store.

The computer and your phone should be on the same wireless network. For example one of the DTU wireless networks.

Now start the DroidCam application on your phone. It should now show an web-address, for example `http://192.168.1.120:4747/video`

Use this address, in the program:

```
use_droid_cam = True
if use_droid_cam:
    url = "http://192.168.1.120:4747/video"
cap = cv2.VideoCapture(url)
```

You should now see the video from your mobile phone on your computer screen. Remember you phone should be unlocked when streaming video.

## Notes on OpenCV and MacOS

Viktor Holmenlund Larsen had problems using MacOS (Big Sur 11.2.3) and fixed it by changing `cv.VideoCapture(0)` to `cv.VideoCapture(1)`. Also by updating `Gstreamer` and its dependencies and deinstalling and resinstalling OpenCV after.

## 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:

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

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 $v_{\text{max}}$ and $v_{\text{min}}$ 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 $\gamma$-mapping of pixel values:

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

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 $\gamma$, 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.

# Exercise 4 - Image Filtering

The purpose of this exercise is to illustrate different image filtering techniques.

## Learning Objectives

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

1. Compute the correlation between an image and a filter using the `scipy.ndimage.correlate` function.
2. Use different border handling strategies when using filtering an image, including `constant` and `reflection`.
3. Implement and apply a mean filter to an image.
4. Implement and apply a median filter to an image (`skimage.filters.median`).
5. Implement and apply a Gaussian filter to an image (`skimage.filters.gaussian`)
6. Describe the effects of applying the mean, the Gaussian and the median filter to images containing Gaussian and outlier noise.
7. Describe the concept on an image edge.
8. Describe the concept of image gradients.
9. Use the Prewitt filter to extract horizontal and vertical edges and their combined magnitude (`skimage.filters.prewitt_h`, `skimage.filters.prewitt_v`, `skimage.filters.prewitt`).
10. Estimate a threshold in an edge image to create a binary image reflecting the significant edges in an image.
11. Implement, test, adapt and evaluate a function that can automatically detect important edges in an image.
12. Implement and test a program that apply filters to a video stream.
13. Test the impact of a video processing frame rate when applying different filters to the video stream.

## Installing Python packages

In this exercise, we will be using both scikit-image, OpenCV and SciPy. You should have these 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/ex4-ImageFiltering/data/)

# Filtering using Python

scikit-image and SciPy contain a large number of image filtering functions. In this exercise, we will explore some of the fundamental functions and touch upon more advanced filters as well.

## Filtering using correlation

We will start by exploring the basic correlation operator from SciPy. Start by importing:

```python
from scipy.ndimage import correlate
```

Now create a small and simple image:

```python
input_img = np.arange(25).reshape(5, 5)
print(input_img)
```

and a simple filter:

```python
weights = [[0, 1, 0],
           [1, 2, 1],
           [0, 1, 0]]
```

Now we can correlate the image with the weights:

```python
res_img = correlate(input_img, weights)
```

### Exercise 1

Print the value in position (3, 3) in `res_img`. Explain the value?

## Border handling

When the value of an output pixel at the boundary of the image is computed, a portion of the filter is usually outside the edge of the input image. One way to handle this, is to assume that the value of the *off-the-edge pixels* of the image are 0. This is called zero padding. Since 0 is the value of a black pixel, the output image will have a dark edge. Another approach is to *reflect* the actual pixel values of the image to the *off-the-edge-pixel*. This is the default behaviour of `correlate`. We can also set the *off-the-edge-pixel* to have a constant value (for example 10) by:

```python
res_img = correlate(input_img, weights, mode="constant", cval=10)
```

### Exercise 2

Compare the output images when using `reflection` and `constant` for the border. Where and why do you see the differences.

## Mean filtering

Now we will try some filters on an artificial image with different types of noise starting with the mean filter.

### Exercise 3

Read and show the image **Gaussian.png** from the exercise material.

Create a mean filter with normalized weights:

```
size = 5
# Two dimensional filter filled with 1
weights = np.ones([size, size])
# Normalize weights
weights = weights / np.sum(weights)
```

Use `correlate` with the **Gaussian.png** image and the mean filter. Show the resulting image together with the input image. What do you observe?

Try to change the size of the filter to 10, 20, 40 etc.. What do you see?

What happens to the noise and what happens to the places in image where there are transitions from light to dark areas?

## Median filtering

The median filter belongs to the group of *rank filters* where the pixel values in a given area are sorted by value and then one of the values are picked. Here the median value of the sorted values.

Start by importing the filter:

```
from skimage.filters import median
```

We can create a *footprint* which marks the size of the median filter and do the filtering like this:

```
size = 5
footprint = np.ones([size, size])
med_img = median(im_org, footprint)
```

### Exercise 4

Filter the **Gaussian.png** image with the median filter with different size (5, 10, 20...). What do you observe? What happens with the noise and with the lighth-dark transitions?

## Comparing mean and median filtering

Try to load and show the **SaltPepper.png** image. This image has noise consist of very dark or very light pixels.

### Exercise 5

Try to use your mean and median filter with different filter sizes on the **Salt-Pepper.png**. What do you observe? Can they remove the noise and what happens to the image?

## Gaussian filter

Scikit-image contains many different filters.

The Gaussian filter is widely used in image processing. It is a smoothing filter that removes high frequencies from the image.

### Exercise 6

Let us try the Gaussian filter on the **Gaussian.png** image. Start by importing the filter:

```python
from skimage.filters import gaussian
```

and do the filtering:

```python
sigma = 1
gauss_img = gaussian(im_org, sigma)
```

Try to change the `sigma` value and observe the result.

### Exercise 7

Use one of your images (or use the **car.png** image) to try the above filters. Especially, try with large filter kernels (larger than 10) with the median and the Gaussian filter. Remember to transform your image into gray-scale before filtering.

What is the visual difference between in the output? Try to observe places where there is clear light-dark transition.

## Edge filters

In image analysis, an *edge* is where there is a large transition from light pixels to dark pixels. It means that there is a *high pixel value gradient* at an edge. Since objects in an image are often of a different color than the background, the outline of the object can sometimes be found where there are edges in the image. It is therefore interesting to apply filters that can estimate the gradients in the image and using them to detect edges.

The **Prewitt filter** is a simple gradient estimation filter. The Python version of the Prewitt filter can estimate the horizontal gradient using the `prewitt_h` filter, the vertical gradient with the `prewitt_v` filter and the *magnitude of the edges* using the `prewitt` filter. The magnitude is computed as

$$V(x,y) = \sqrt{(P_v^2 + P_h^2)} \ ,$$

where $P_v$ and $P_h$ are the outputs of the vertical and horizontal Prewitt filters.

Start by importing the filter:

```
from skimage.filters import prewitt_h
from skimage.filters import prewitt_v
from skimage.filters import prewitt
```

### Exercise 8

Try to filter the **donald_1.png** photo with the `prewitt_h` and `prewitt_v` filters and show the output without converting the output to unsigned byte. Notice that the output range is [-1, 1]. Try to explain what features of the image that gets high and low values when using the two filters?

### Exercise 9

Use the `prewitt` filter on **donald_1.png**. What do you see?

## Edge detection in medical images

The **ElbowCTSlice.png** image is one slice of a CT scan of an elbow from a person that climbed, wanted to show off, fell, landed on his arm and fractured a bone.

### Exercise 10

The goal of this exercise is to detect the edges that seperates the bone from the soft tissue and the edges that separates the elbow from the background. Your detection algorithm should follow this outline:

- Read the CT image
- Filter the image using either a Gaussian filter or a median filter
- Compute the gradients in the filtered image using a Prewitt filter
- Use Otsu's thresholding method to compute a threshold, T, in the gradient image
- Apply the threshold, T, to the gradient image to create a binary image.

The final binary should contain the edges we are looking for. It will probably contain noise as well. We will explore methods to remove this noise later in the course.

You should experiment and find out:

- Does the median or Gaussian filter give the best result?
- Should you use both the median and the Gaussian filter?

- What filter size gives the best result?
- What sigma in the Gaussian filter gives the best result?

**Tip:** To get a better understanding of your output, uou can use the scaled visualization and colormapping that we explored in an earlier exercise:

```
min_val = edge_img.min()
max_val = edge_img.max()
io.imshow(edge_img, vmin=min_val, vmax=max_val, cmap="terrain")
```

## Video filtering

Now try to make a small program, that acquires video from your webcam/telephone, filters it and shows the filtered output. In the exercise material there is a program that can be modified.

### Exercise 11

Modify the `process_gray_image` function in the program so it performs a Prewitt filter on the input image.

Also try to make it perform the automatic edge-detection (Prewitt + Otsu) from exercise 10.

### Exercise 12

Try to use a median filter with a size of 10 on the video stream. What happens with the frames-per-second? Why?

## References

- sci-kit image filters
- rank filters
- scipy correlate

## Differences between `matplotlib.pyplot.imshow()` and `skimage.io.imshow()`

You should also be aware of the differences in the default behaviour between scikit-image and matplotlib when displaying an image. The default behaviour of matplotlib.pyplot.imshow() is to use the dynamic range of the image values to plot the image. It is, if the image intensities range from [26, 173], the black colour is assigned to 26 and the white to 173. Meanwhile, skimage.io.imshow() displays the dynamic range according to the image type, [0, 255] for integer images and [0., 1.] for float images.

Feel free to use either package, as far as you know what you are doing.

# Exercise 4b - Image Morphology

The purpose of this exercise is to implement, test and validate different approaches to binary image morphological operations.

## Learning Objectives

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

1. Define a *structuring element* (also called a *footprint*) using the `disk` function from the `skimage.morphology` package.
2. Perform the morphological operations: *erosion*, *dilation*, *opening* and *closing* on binary images.
3. Compute the outlines seen in a binary image.
4. Use morphological operations to remove holes in objects.
5. Use morphological operations to separate binary objects.
6. Select appropriate footprints based on image properties and object appearance.
7. Combine morphological operations to clean and separate objects.

## Installing Python packages

In this exercise, we will be using scikit-image. You should have this library 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/ex4b-ImageMorphology/data/)

## Image Morphology in Python

scikit-image contain a variety of morphological operations. In this exercise we will explore the use of some of these operations on binary image.

Start by importing some function:

```
from skimage.morphology import erosion, dilation, opening, closing
from skimage.morphology import disk
```

and define a convenience function to show two images side by side:

```
# From https://scikit-image.org/docs/stable/auto_examples/applications/plot_morphology.html
def plot_comparison(original, filtered, filter_name):
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4), sharex=True,
                                   sharey=True)
    ax1.imshow(original, cmap=plt.cm.gray)
    ax1.set_title('original')
    ax1.axis('off')
    ax2.imshow(filtered, cmap=plt.cm.gray)
    ax2.set_title(filter_name)
    ax2.axis('off')
    io.show()
```

## Image morphology on a single object

An image, **lego_5.png** of a lego brick can be used to test some of the basic functions.

### Exercise 1

We will start by computing a binary image from the lego image:

- Read the image into **im_org**.
- Convert the image to gray scale.
- Find a threshold using *Otsu's method*.
- Apply the treshold and generate a binary image **bin_img**.
- Visualize the image using `plot_comparison(im_org, bin_img, 'Binary image')`

As ncan be seen, the lego brick is not *segmented* perfectly. There are holes in the segmentation. Let us see if what we can do.

### Exercise 2

We will start by creating a *structuring element*. In scikit-image they are called *footprint*. A disk shaped footprint can be created by:

```
footprint = disk(2)
# Check the size and shape of the structuring element
print(footprint)
```

The morphological operation **erosion** can remove small objects, separate objects and make objects smaller. Try it on the binary lego image:

```
eroded = erosion(bin_img, footprint)
plot_comparison(bin_img, eroded, 'erosion')
```

Experiement with different sizes of the footprint and observe the results.

### Exercise 3

The morphological operation **dilation** makes objects larger, closes holes and connects objects. Try it on the binary lego image:

```
dilated = dilation(bin_img, footprint)
plot_comparison(bin_img, dilated, 'dilation')
```

Experiement with different sizes of the footprint and observe the results.

### Exercise 4

The morphological operation **opening** removes small objects without changing the size of the remaining objects. Try it on the binary lego image:

```
opened = opening(bin_img, footprint)
plot_comparison(bin_img, opened, 'opening')
```

Experiement with different sizes of the footprint and observe the results.

### Exercise 5

The morphological operation **closing** closes holes in objects without changing the size of the remaining objects. Try it on the binary lego image:

```
closed = closing(bin_img, footprint)
plot_comparison(bin_img, closed, 'closing')
```

Experiement with different sizes of the footprint and observe the results.

## Object outline

It can be useful to compute the outline of an object both to measure the perimeter but also to see if it contains holes or other types of noise. Start by defining an outline function:

```
def compute_outline(bin_img):
    """
    Computes the outline of a binary image
    """
    footprint = disk(1)
    dilated = dilation(bin_img, footprint)
    outline = np.logical_xor(dilated, bin_img)
    return outline
```

### Exercise 6

Compute the outline of the binary image of the lego brick. What do you observe?

### Exercise 7

Try the following:

- Do an *opening* with a disk of size 1 on the binary lego image.
- Do a *closing* with a disk of size 15 on the result of the opening.

- Compute the outline and visualize it.

What do you observe and why does the result look like that?

## Morphology on multiple objects

Let us try to do some analysis on images with multiple objects.

### Exercise 8

Start by: - reading the **lego_7.png** image and convert it to gray scale. - Compute a treshold using *Otsu's method* and apply it to the image. - Show the binary image together with the original. - Compute the outline of the binary image and show it with the binary image.

What do you observe?

### Exercise 9

We would like to find a way so only the outline of the entire brick is computed. So for each lego brick there should only be one closed curve.

Try using the *closing* operations and find out which size of footprint that gives the desired result?

### Exercise 10

Try the above on the **lego_3.png** image. What do you observe?

## Morphology on multiple connected objects

Morphology is a strong tool that can be used to clean images and separate connected objects. In image **lego_9.png** some lego bricks are touching. We would like to see if we can separate them.

### Exercise 11

Start by: - reading the **lego_9.png** image and convert it to gray scale. - Compute a treshold using *Otsu's method* and apply it to the image. - Show the binary image together with the original. - Compute the outline of the binary image and show it with the binary image.

What do you observe?

### Exercise 12

Let us start by trying to remove the noise holes inside the lego bricks. Do that with an *closing* and find a good footprint size. Compute the outline and see what you observe?

### Exercise 13

Now we will try to separate the objects. Try using a *erosion* on the image that you repaired in exercise 12. You should probably use a rather large footprint. How large does it need to be in order to split the objects?

### Exercise 14

The objects lost a lot of size in the previous step. Try to use *dilate* to make them larger. How large can you make them before they start touching?

## Puzzle piece analysis

We would like to make a program that can help solving puzzles. The first task is to outline each piece. A photo, **puzzle_pieces.png** is provided.

### Exercise 15

Use the previosly used methods to compute a binary image from the puzzle photo. What do you observe?

### Exercise 16

Try to use a an *opening* with a large footprint to clean the binary. Compute the outline. Do we have good outlines for all the pieces?

The conclusion is that you can solve a lot of problems using morphological operations but sometimes it is better to think even more about how to acquire the images.

## References

- sci-kit image morphology
- sci-kit morphology examples

## Exercise 5 - BLOB Analysis (connected component analysis and object classification)

The purpose of this exercise is to implement, test and validate connected component analysis methods. Also known as BLOB (binary large object) analysis.

The methods will be used to create a small program that can count cell nuclei.

## Learning Objectives

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

1. Preprocess a colour image so it is suitable for BLOB analysis using color to gray transformations and threshold selection.
2. Use slicing to extract regions of an image for further analysis.
3. Use `segmentation.clear_border` to remove border BLOBs.
4. Apply suitable morphological operations to remove small BLOBs, close holes and generally make a binary image suitable for BLOB analysis.
5. Use `measure.label` to create labels from a binary image.
6. Visualize labels using `label2rgb`.
7. Compute BLOB features using `measure.regionprops` including BLOB area and perimeter.
8. Remove BLOBs that have certain features.
9. Extract BLOB features and plot feature spaces as for example area versus perimeter and area versus circularity.
10. Choose a set of BLOB features that separates objects from noise.
11. Implement and test a small program for cell nuclei classification and counting.

## Installing Python packages

In this exercise, we will be using scikit-image. You should have this library 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/tree/main/exercises/ex5-BLOBAnalysis/data)

## BLOB Analysis in Python

Start by importing some function:

```
from skimage import io, color, morphology
from skimage.util import img_as_float, img_as_ubyte
import matplotlib.pyplot as plt
import numpy as np
import math
from skimage.filters import threshold_otsu
from skimage import segmentation
from skimage import measure
from skimage.color import label2rgb
```

and define a convenience function to show two images side by side:

```
def show_comparison(original, modified, modified_name):
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4), sharex=True,
                                   sharey=True)
    ax1.imshow(original)
    ax1.set_title('Original')
    ax1.axis('off')
    ax2.imshow(modified)
    ax2.set_title(modified_name)
    ax2.axis('off')
    io.show()
```

## LEGO Classification

We will start by trying some BLOB analysis approaches on a photo of some Lego bricks: **lego_4_small.png**.

### Exercise 1: Binary image from original image

Read the image, convert it to grayscale and use *Otsus* method to compute and apply a threshold.

Show the binary image together with the original image.

### Exercise 2: Remove border BLOBs

Use `segmentation.clear_border` to remove border pixels from the binary image.

### Exercise 3: Cleaning using morphological operations

In order to remove remove noise and close holes, you should do a morphological closing followed by a morphological opening with a disk shaped structuring element with radius 5. See Exercise 4b if you are in doubt.

### Exercise 4: Find labels

The actual connected component analysis / BLOB analysis is performed using `measure.label` :

```
label_img = measure.label(img_open)
n_labels = label_img.max()
print(f"Number of labels: {n_labels}")
```

### Exercise 5: Visualize found labels

We can use the function `label2rbg` to create a visualization of the found BLOBS. Show this together with the original image.

### Exericse 6: Compute BLOB features

It is possible to compute a wide variety of BLOB features using the `measure.regionprops` function:

```
region_props = measure.regionprops(label_img)
areas = np.array([prop.area for prop in region_props])
plt.hist(areas, bins=50)
plt.show()
```

### Exercise 7: Exploring BLOB features

There is an example program called `Ex5-BlobAnalysisInteractive.py` in the exercise material folder.

With that program, you can explore different BLOB features interactively. It requires installation of `plotly`:

```
conda install -c plotly plotly=5.10.0
```

## Cell counting

The goal of this part of the exercise, is to create a small program that can automatically count the number of cell nuclei in an image.

The images used for the exercise is acquired by the Danish company Chemometec using their image-based cytometers. A cytometer is a machine used in many laboratories to do automated cell counting and analysis. An example image can be seen in below where U2OS cells (human bone cells) have been imaged using ultraviolet (UV) microscopy and a fluorescent staining method named DAPI. Using DAPI staining only the cell nuclei are visible which makes the method very suitable for cell counting.

UV DAPI **U2OS cells**: To the left image acquired using UV microscopy and to the right the corresponding DAPI image.

The raw images from the Cytometer are 1920x1440 pixels and each pixel is 16 bit (values from 0 to 65535). The resolution is 1.11 $\mu m$ / pixel.

To make it easier to develop the cell counting program we start by working with smaller areas of the raw images. The images are also converted to 8 bit grayscale images:

```
in_dir = "data/"
img_org = io.imread(in_dir + 'Sample E2 - U2OS DAPI channel.tiff')
# slice to extract smaller image
img_small = img_org[700:1200, 900:1400]
img_gray = img_as_ubyte(img_small)
io.imshow(img_gray, vmin=0, vmax=150)
plt.title('DAPI Stained U2OS cell nuclei')
io.show()
```

As can be seen we use *slicing* to extract a part of the image. You can use `vmin` and `vmax` to visualise specific gray scale ranges (0 to 150 in the example above). Adjust these limits to find out where the cell nuclei are most visible.

Initially, we would like to apply a threshold to create a binary image where nuclei are foreground. To select a good threshold, inspect the histogram:

```
# avoid bin with value 0 due to the very large number of background pixels
plt.hist(img_gray.ravel(), bins=256, range=(1, 100))
io.show()
```

### Exercise 8: Threshold selection

Select an appropriate threshold, that seperates nuclei from the background. You can set it manually or use *Otsus* method.

Show the binary image together with the original image and evaluate if you got the information you wanted in the binary image.

It can be seen that there is some noise (non-nuclei) present and that some nuclei are connected. Nuclei that are overlapping very much should be discarded in the analysis. However, if they are only touching each other a little we can try to separate them. More on this later.

To make the following analysis easier the objects that touches the border should be removed.

### Exercise 9: Remove border BLOBS

Use `segmentation.clear_border` to remove border pixels from the binary image.

To be able to analyse the individual objects, the objects should be labelled.

```
label_img = measure.label(img_c_b)
image_label_overlay = label2rgb(label_img)
show_comparison(img_org, image_label_overlay, 'Found BLOBS')
```

In this image, each object has a separate color - does it look reasonable?

### Exercise 10: BLOB features

The task is now to find some **object features** that identify the cell nuclei and let us remove noise and connected nuclei. We use the function `regionprops` to compute a set of features for each object:

```
region_props = measure.regionprops(label_img)
```

For example can the area of the first object be seen by: `print(region_props[0].area)`.

A quick way to gather all areas:

```
areas = np.array([prop.area for prop in region_props])
```

We can try if the area of the objects is enough to remove invalid object. Plot a histogram of all the areas and see if it can be used to identify well separated nuclei from overlapping nuclei and noise. You should probably play around with the number of bins in your histogram plotting function.

### Exercise 11: BLOB classification by area

Select a minimum and maximum allowed area and use the following to visualise the result:

```
min_area =
max_area =

# Create a copy of the label_img
label_img_filter = label_img
for region in region_props:
    # Find the areas that do not fit our criteria
    if region.area > max_area or region.area < min_area:
        # set the pixels in the invalid areas to background
        for cords in region.coords:
            label_img_filter[cords[0], cords[1]] = 0
# Create binary image from the filtered label image
i_area = label_img_filter > 0
show_comparison(img_small, i_area, 'Found nuclei based on area')
```

Can you find an area interval that works well for these nuclei?

### Exercise 12: Feature space

Extract all the perimeters of the BLOBS:

```
perimeters = np.array([prop.perimeter for prop in region_props])
```

Try to plot the areas versus the perimeters.

### Exercise 13: BLOB Circularity

We should also examine if the shape of the cells can identify them. A good measure of how circular an object is can be computed as:

$$f_{\text{circ}} = \frac{4\pi A}{P^2},$$

where $A$ is the object area and $P$ is the perimeter. A circle has a circularity close to 1, and very-non-circular object have circularity close to 0.

Compute the circularity for all objects and plot a histogram.

Select some appropriate ranges of accepted circularity. Use these ranges to select only the cells with acceptable areas and circularity and show them in an image.

### Exercise 14: BLOB circularity and area

Try to plot the areas versus the circularity. What do you observe?

Extend your method to return the number (the count) of well-formed nuclei in the image.

### Exercise 15: large scale testing

Try to test the method on a larger set of training images. Use slicing to select the different regions from the raw image.

### Exercise 16: COS7 cell classification

Try your method on the **Sample G1 - COS7 cells DAPI channel.tiff** image. COS7 cells are African Green Monkey Fibroblast-like Kidney Cells used for a variety of research purposes.

### Exercise 17: Handling overlap

In certain cases cell nuclei are touching and are therefore being treated as one object. It can sometimes be solved using for example the morphological operation **opening** before the object labelling. The operation **erosion** can also be used but it changes the object area.

## References

- sci-kit image label
- sci-kit image region properties
- Measure region properties

# Exercise 6 - Pixel classification and object segmentation

In the first part of this exercise, we will use pixel classification to label pixels in an image. In the second part, pixel classification will be combined with BLOB analysis to segment the spleen from a computed tomography (CT) scan.

## Learning Objectives

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

1. Describe the basic anatomy of a abdominal computed tomography scan including the liver, the spleen, the kidneys, bone and fat.
2. Describe the concept of **Hounsfield units** as used in computed tomography scans.
3. Describe the relationship between the Hounsfield unit corresponding to water and to air.
4. Use pixel value mapping in `io.imshow` to get optimal contrast for 16-bit medical scans.
5. Use a binary segmentation mask to extract pixel values corresponding to the pixels covered by the mask.
6. Compute standard measures as the average value and the standard deviation of a selected set of pixel values.
7. Visualize the histogram of a selected set of pixel values.
8. Use the SciPy function `norm.pdf` to sample values in a Gaussian distribution with a given mean and standard deviation.
9. Plot a histogram of a selected set of pixel values together with the best fitting Gaussian distribution.
10. Visualize and evaluate the class overlap by plotting fitted Gaussian functions of each pre-defined class.
11. Describe the concept of *minimum distance classification.*
12. Compute class ranges using the concept of *minimum distance classification.*
13. Apply a *minimum distance classifier* to an image and visualize the results.
14. Visually evaluate the result of a pixel classification by visually comparing with a ground truth image.
15. Compute the class ranges in a *parametric classifier* by visually inspecting the Gaussians representing each class and manually finding where they cross.
16. Use `norm.pdf` to find the class with the highest probability given a pixel value.
17. Use `norm.pdf` to compute the class ranges by testing the probabilities with a set of pixel values.
18. Apply a *parametric classifier* to an image and visualize the results.
19. Use morphological opening and closing to repair holes in objects and separate objects in a binary image.

20. Use BLOB analysis to label objects in a binary image.
21. Use BLOB feature based classification to identify an object in an image. For example the spleen in a computed tomography scan.
22. Describe the concept of the **DICE score**.
23. Compute the DICE score between two segmentations.
24. Compute and evaluate the DICE score between a computed segmentation and a ground truth segmentation.
25. Evaluate and optimize a segmentation algorithm based on visual results and DICE scores.
26. Describe why it is important to split data into a training set, a validation set and a test set.
27. Compute the final result of an algorithm on a test set and evaluate the results both visually and using the DICE score.

## Installing Python packages

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

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

Let us start with some imports and defining a convenience function:

```python
from skimage import io, color
from skimage.morphology import binary_closing, binary_opening
from skimage.morphology import disk
import matplotlib.pyplot as plt
import numpy as np
from skimage import measure
from skimage.color import label2rgb
import pydicom as dicom
from scipy.stats import norm
from scipy.spatial import distance


def show_comparison(original, modified, modified_name):
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4), sharex=True,
                                   sharey=True)
    ax1.imshow(original, cmap="gray", vmin=-200, vmax=500)
    ax1.set_title('Original')
    ax1.axis('off')
    ax2.imshow(modified)
    ax2.set_title(modified_name)
    ax2.axis('off')
    io.show()
```

## Exercise data and material

The data and material needed for this exercise can be found here: exercise data and material (https://github.com/RasmusRPaulsen/DTUImageAnalysis/tree/main/exercises/ex6-PixelClassificationAndObjectSegmentation/data)

There are one training image, three validation images and three test images. They have ground truth annotations of the spleen that we will use for training, validation and testing of our algorithm.

## Abdominal computed tomography

The images in this exercise are DICOM images from a computed tomography (CT) scan of the abdominal area. An example can be seen below, where the anatomies we are working with are marked. You should

A CT scan is normally a 3D volume with many slices, but in this exercise, we will only work with one slice at a time. We therefore call one *slice* for an image.

In this exercise, we will mostly focus on the **spleen** but also examine the **liver, kidneys, fat tissue and bone.**

Abdominal scan with labels

### Hounsfield units

The pixels in the images are stored as 16-bit integers, meaning their values can be in the range of $[-32.768, 32.767]$. In a CT image, the values are represented as **Hounsfield units** (HU). Hounsfield units are used in computed tomography to characterise the X-ray absorption of different tissues. A CT scanner is normally calibrated so a pixel with Hounsfield unit 0 has an absorbance equal to water and a pixel with Hounsfield unit -1000 has absorbance equal to air. Bone absorbs a lot of radiation and therefore have high HU values (300-800) and fat absorbs less radiation than water and has HU units around -100. Several organs have similar HU values since the soft-tissue composition of the organs have similar X-ray absorption. In the figure below (from Erich Krestel, “Imaging Systems for Medical Diagnostics”, 1990, Siemens) some typical HU units for organs can be seen. They are, however, not always consistent from scanner to scanner and hospital to hospital.

Abdominal scan with labels

## Explorative analysis of one CT scan

Let us start by examining one of the CT scan slices from the training set. You can read the first slice like this:

```
in_dir = "data/"
ct = dicom.read_file(in_dir + 'Training.dcm')
img = ct.pixel_array
```

```
print(img.shape)
print(img.dtype)
```

You should visualise the slice, so the organs of interest have a suitable brigthness and contrast. One way is to manipulate the minimum and maximum values proviede to `imshow`.

**Exercise 1**: *The spleen typically has HU units in the range of 0 to 150. Try to make a good visualization of the CT scan and spleen using (replace the question marks with values):*

```
io.imshow(img, vmin=?, vmax=?, cmap='gray')
io.show()
```

An expert has provided annotations of **bone, fat, kidneys, liver and spleen**. They are stored as *mask* files which is an image with the same size as the input image, where the annotated pixels are 1 and the rest are 0. They are found as **BoneROI.png, FatROI.png, KidneyROI.png, LiverROI and SpleenROI.png**.

You can use the original image and a mask to get the values of the pixels inside the mask:

```
spleen_roi = io.imread(in_dir + 'SpleenROI.png')
# convert to boolean image
spleen_mask = spleen_roi > 0
spleen_values = img[spleen_mask]
```

**Exercise 2**: *Compute the average and standard deviation of the Hounsfield units found in the spleen in the training image. Do they correspond to the values found in the above figure?*

**Exercise 3**: *Plot a histogram of the pixel values of the spleen. Does it look like they are Gaussian distributed?*

The function `norm.pdf` from `SciPy` represents a Gaussian probability density function (PDF). It can for example be used to plot a Gaussian distribution with a given mean and standard deviation.

This can be used to create a fitted Gaussian distribution of the spleen values:

```
n, bins, patches = plt.hist(spleen_values, 60, density=1)
pdf_spleen = norm.pdf(bins, mu_spleen, std_spleen)
plt.plot(bins, pdf_spleen)
plt.xlabel('Hounsfield unit')
plt.ylabel('Frequency')
plt.title('Spleen values in CT scan')
plt.show()
```

Here `mu_spleen` and `std_spleen` are the average and standard deviation of the spleen values.

**Exercise 4**: *Plot histograms and their fitted Gaussians of several of the tissues types. Do they all look like they are Gaussian distributed?*

The fitted Gaussians are good for inspecting class separation and how much the class overlap. Plotting several fitted Gaussians can for example be done like this:

```
# Hounsfield unit limits of the plot
min_hu = -200
max_hu = 1000
hu_range = np.arange(min_hu, max_hu, 1.0)
pdf_spleen = norm.pdf(hu_range, mu_spleen, std_spleen)
pdf_bone = norm.pdf(hu_range, mu_bone, std_bone)
plt.plot(hu_range, pdf_spleen, 'r--', label="spleen")
plt.plot(hu_range, pdf_bone, 'g', label="bone")
plt.title("Fitted Gaussians")
plt.legend()
plt.show()
```

**Exercise 5**: *Plot the fitted Gaussians of bone, fat, kidneys, liver and spleen. What classes are easy to seperate and which classes are hard to seperate?*

**Exercise 6**: *Define the classes that we aim at classifying. Perhaps some classes should be combined into one class?*

## Minimum distance pixel classification

In the **minimum distance classifier** the pixel value class ranges are defined using the average values of the training values. If you have two classes, the threshold between them is defined as the mid-point between the two class value averages.

In the following, we will define four classes: **background, fat, soft tissue and bone**, where soft-tissue is a combination of the values of the spleen, liver and kidneys. We manually set the threshold for background to -200. So all pixels below -200 are set to background.

**Exercise 7**: *Compute the class ranges defining fat, soft tissue and bone.*

You can now use:

```
t_background = -200
fat_img = (img > t_background) & (img <= t_fat_soft)
```

to create an image where all the pixel that are classified as fat, will be 1 and the rest 0. Here `t_fat_soft` is the threshold between the fat and the soft tissue class.

**Exercise 8**: *Create class images: fat_img, soft_img and bone_img representing the fat, soft tissue and bone found in the image.*

To visualize the classification results you can use:

```
label_img = fat_img + 2 * soft_img + 3 * bone_img
image_label_overlay = label2rgb(label_img)
show_comparison(img, image_label_overlay, 'Classification result')
```

**Exercise 9**: *Visualize your classification result and compare it to the anatomical image in the start of the exercise. Does your results look plausible?*

## Parametric pixel classification

In the **parametric classifier**, the standard deviation of the training pixel values is also used when determinin the class ranges. In the following, we are also trying to classify **background, fat, soft tissue and bone**.

We start by finding the class ranges by manually inspecting the fitted Gaussians from each class.

As in the last exercise, we can still se the background-fat threshold to be -200.

**Exercise 9**: *Plot the fitted Gaussians of the training values and manually find the intersection between the curves.*

**Exercise 10**: *Use the same technique as in exercise 7, 8 and 9 to visualize your classification results. Did it change compared to the minimum distance classifier?*

An alternative way of finding the class ranges is to test which class has a the highest probability for a given value. The `norm.pdf` function can be used for that. For example:

```
if norm.pdf(test_value, mu_soft, std_soft) > norm.pdf(test_value, mu_bone, std_bone):
    print(f"For value {test_value} the class is soft tissue")
else:
    print(f"For value {test_value} the class is bone")
```

here the `test_value` is a pixel value that you want to assign a class. One way to use this is to create a *look-up-table* where for each possible HU unit (for example 100, 101, 102 etc), the most probably class is noted. Doing this will give you the pixel value, where the two neighbouring classes meet.

**Exercise 11**: *Use `norm.pdf` to find the optimal class ranges between fat, soft tissue and bone.*

## Object segmentation - The spleen finder

The goal of this part of the exercise, is to create a program that can automatically segment the spleen in CT images.

We start by using the **Training.dcm** image and the expert provided annotations.

**Exercise 11**: *Inspect the values of the spleen as in exercise 3 and select a lower and upper threshold to create a spleen class range.*

You can now use:

```
spleen_estimate = (img > t_1) & (img < t_2)
spleen_label_colour = color.label2rgb(spleen_estimate)
io.imshow(spleen_label_colour)
plt.title("First spleen estimate")
io.show()
```

to show your first spleen estimate. As can be seen, there a many non-spleen areas in the result. The spleen is also connected to another anatomy.

Luckily, we can use morphological operations to fix these issues:

```
footprint = disk(?)
closed = binary_closing(spleen_estimate, footprint)

footprint = disk(?)
opened = binary_opening(closed, footprint)
```

**Exercise 12**: *Use the above morphological operations to seperate the spleen from other organs and close holes. Change the values where there are question marks to change the size of the used structuring elements.*

Now we can use BLOB analysis to do a feature based classification of the spleen.

**Exercise 12**: *Use the methods from BLOB analysis to compute BLOB features for every seperated BLOB in the image. You can for example start by:*

```
label_img = measure.label(opened)
```

**Exercise 13**: *Inspect the labeled image and validate the success of separating the spleen from the other objects. If it is connected (have the same color) to another organ, you should experiment with the kernel sizes in the morphological operations.*

To be able to keep only the spleen we need to find out which BLOB features, that is special for the spleen. By using `measure.regionprops` many different BLOB features can be computed, including area and perimeter.

You can for example use:

```
min_area = ?
max_area = ?

# Create a copy of the label_img
label_img_filter = label_img.copy()
for region in region_props:
    # Find the areas that do not fit our criteria
    if region.area > max_area or region.area < min_area:
        # set the pixels in the invalid areas to background
        for cords in region.coords:
            label_img_filter[cords[0], cords[1]] = 0
# Create binary image from the filtered label image
i_area = label_img_filter > 0
show_comparison(img, i_area, 'Found spleen based on area')
```

to create a filtered binary image, where only valid BLOBs are remaining.

**Exercise 14**: *Extend the method above to include several BLOB features. For example area and perimeter. Find the combination of features and feature value limits that will result in only the spleen remaining.*

**Exercise 15**: *Create a function `spleen_finder(img)` that takes as input a CT image and returns a binary image, where the pixels with value 1 represent the spleen and the pixels with value 0 everything else.*

**Exercise 16**: *Test your function on the images called* ***Validation1.dcm***, ***Validation2.dcm*** *and* ***Validation3.dcm***. *Do you succeed in finding the spleen in all the validation images?*

## DICE Score

We would like evaluate how good we are at finding the spleen by comparing our found spleen with ground truth annotations of the spleen. The **DICE score** (also called the DICE coefficient or the DICE distance) is a standard method of comparing one segmentation with another segmentation.

If segmentation one is called `X` and the second segmentation called `Y`. The DICE score is computed as:

$$\text{DICE} = \frac{2|X \cap Y|}{|X| + |Y|}$$

where $|X \cap Y|$ is the area (in pixels) of the overlap of the two segmentations and is $|X| + |Y|$ the area of the union of the two segmentation. This can be visualized as:

DICE Score

The DICE score is one if there is a perfect overlap between the two segmentations and zero if there is no overlap at all. A DICE score above 0.95 means that the two segmentations are very similar.

Using SciPy we can compute the DICE score as:

```
ground_truth_img = io.imread(in_dir + 'Validation1_spleen.png')
gt_bin = ground_truth_img > 0
dice_score = 1 - distance.dice(i_area.ravel(), gt_bin.ravel())
print(f"DICE score {dice_score}")
```

**Exercise 17**: *Compute the DICE score for your found spleen segmentations compared to the ground truth segmentations for the three validation images. How high DICE scores do you achieve?*

## Testing on an independent test set

*Overfitting* occurs when an algorithm has been developed on a training set and has become so specific to that set of data, that it works badly on other similar data. To avoid this, it is necessary to test an algorithm on an independent test set. We have provided three test images **Test1.dcm**, **Test2.dcm** and **Test3.dcm** with ground truth spleen annotations.

**Exercise 18**: *Use your spleen finder program to find the spleen on the three test images and compute the DICE score. What is the result of your independent test?*

## References

- Normal distribution
- DICE dissimilarity

# Image Analysis
# Exercise - Advanced segmentation
# Fisherman's Linear discriminant analysis for segmentation

## Introduction

The exercise is to the extent of the pixel-wise classification problem from being based on the intensity histogram of a single image modality to combining multiple image modalities. Hence, here we wish to segment image features into two classes by training a classifier based on the intensity information from multiple image modalities.
Multiple-image modalities just mean a series of images that contain different but complementary image information of the same object. It is assumed that the image modalities have the same size, so we have a pixel-to-pixel correspondence between the two images. An image feature is an identifiable object in the image e.g., of a dog, moon rocket, or brain tissue types that we wish to segment into individual classes.

Here we aim to segment two types of brain tissues into two feature classes. To improve the segmentation, we wish to combine two MRI image modalities that contain different intensity information of the feature classes of interest: one is a “T1 weighted MRI” and the other is a “T2 weighted MRI”. Both are acquired at the same time and are overlapping.

Get the data from **courses.compute.dtu.dk/02502**.

Exercise - You simply go step-by-step and fill in the command lines and answer/discuss the questions (Q1-Q12).

## Theory

**The Linear Discriminate Classifier**

As a classifier, we will use a class of linear discriminate functions that aims to place a hyperplane in the multi-dimensional feature space that acts as a decision boundary to segment two features into classes. Since we only look at image intensities of two image modalities our multi-dimensional feature space is a 2D intensity histogram. The linear discriminant classifier is based on the Bayesian theory where the posterior probability is the probability of voxel $\boldsymbol{x}$ belonging to class $\boldsymbol{C_i}$. The voxel x belongs to the class with the highest posterior probability. A class posterior probability is expressed by Bayes:

$$P(Ci|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|\boldsymbol{\mu_i},\boldsymbol{\Sigma_i},)P(C_i)}{P(\boldsymbol{x})} \text{ [Eq. 1]}$$

Where

- $P(x|\mu_i,\Sigma_i,) = K_i \, exp((x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i))$ is the conditional probability. We assume that the distribution of data for a feature can be modeled using a parametric Gaussian model in multiple dimensions i.e.: $\mathcal{N}(\mu_i,\Sigma_i)$
- *$P(C_i)$* is the prior probability for each class and $\sum_i P(\boldsymbol{C_i}) = 1$
- *$P(x)$* is the marginal probability and is $\sum_i P(\boldsymbol{C_i}|x)$.

*In Eq. 1 we need to train the model using training examples representative of the distribution of features to be segmented. The training examples we often draw ourselves or are provided by an expert. Given the training examples, we train the model parameter which are the means, covariances, and prior probabilities for each class.*

The linear discriminant function for two classes is based on the log-ratio between the two posterior probabilities:

$$\text{Log}\left(\frac{P(C2|\boldsymbol{x})}{P(C1|\boldsymbol{x})}\right) > \log(T) \text{ [Eq. 2]}$$

Note, we take the log() on both sides as a trick to make the expressions easier to realize the model practically but then remember: To calculate the final class posterior probabilities once the classifier model has been trained one needs to account for the log(P(Ci|x)).

We derive the model based on the expression in Eq 2 and its model assumptions that define the model. Firstly, we assume homoscedasticity of variances meaning that the variance of the covariance matrix of each feature is the same for all classes ($\Sigma_1 = \Sigma_2 = \Sigma_0$). Further, we assume to have isotropic covariances i.e., no preferred direction in the variance meaning they have round shape distribution in 2D as illustrated in Figure 1. Hence, covariances in the off-diagonal of the covariance matrix are zero and we have equal variances in the diagonal.

With these assumptions, Eq. 2 is a Linear Discriminant Analysis (LDA) classifier and can be expressed by the means and the covariances of the two gaussian distributions and their prior probabilities:

$$\log\left(\frac{P1}{P2}\right) - \frac{1}{2}(\boldsymbol{\mu_2} + \boldsymbol{\mu_1})^T \Sigma_0^{-1}(\boldsymbol{\mu_2} - \boldsymbol{\mu_1}) + \boldsymbol{x}^T \Sigma_0^{-1}(\boldsymbol{\mu_2} - \boldsymbol{\mu_1}) > \log(T) \text{ [Eq. 3]}$$

The expression can look a bit complicated at first glimpse but if we add colors a general structure of the expression appears. The green part $\Sigma_0^{-1}(\boldsymbol{\mu_2} - \boldsymbol{\mu_1})$ we name a weight vector, ***w*** which is normal to the hyperplane, and along ***w*** the decision boundary separating the two classes is to be placed. We can say that ***w*** determines how the hyperplane is orientated in the coordinate system to best separate the two classes. In other words, ***w*** is our model like in Eq. 3. Remember, it is defined by its assumptions and therefore it is not guaranteed that it is the best solution if the data do not agree with the assumptions. The orange part $\log\left(\frac{P1}{P2}\right) - \frac{1}{2}(\boldsymbol{\mu_2} + \boldsymbol{\mu_1})^T$ we call the bias ***c*** which describes where the hyperplane hence the decision boundary along the ***w*** is placed. Basically, the expression for ***c*** says that the decision boundary is placed halfway between the means of the two class distributions weighted with the ratio of the prior probability for each class. So, it is halfway between the class means if the two classes have equal prior probabilities, *P(C$_i$)* (i.e., 0.5). In 1D and in this case, it is the same as a parametric classifier where we define a threshold at the point where the two distributions cross having equal probabilities.

We can reformulate Eq. 3 in a general way:

$$\boldsymbol{y_{C\in 2}}(\boldsymbol{x}) = \boldsymbol{x^T w} + \boldsymbol{w_o} \text{ ; where } \boldsymbol{w_o} = \boldsymbol{c\, w} \text{ [Eq. 4]}$$

The weight vector **w** is multiplied both with the vector of a data point ***x*** and with the vector of the bias ***c.*** Multiplying two vectors is the same as the dot-product between two vectors i.e., $\boldsymbol{ab} = \|\boldsymbol{a}\|\|\boldsymbol{b}\| \cos\theta$. Hence, the function of the weight vector is to rotate the hyperplane in feature space with an angle that best separates the two classes. Therefore, when ***w*** is multiplied with the bias ***c*** it is projected along ***w*** and defines the decision boundary at ***w$_o$***. When multiplied with a data point vector ***x*** it projects the data point along ***w*** and whether the projected data point belongs to class 2 depends on its projected position in relation to the decision boundary. Figure 1 illustrates the LDA model and two gaussian classes with equal isotropic variances - as the model assumption subscribes - the principle of how the weight vector projects the hyperplane and decision boundary independent of the orientation between the two class means and the coordinate system Figure 1 (A vs B).

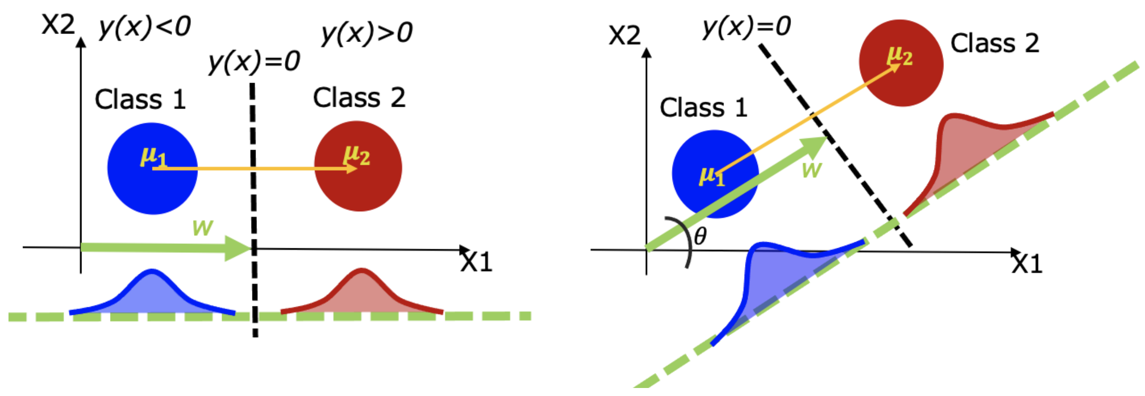

Figure 1 Principle how the LDA model can be explained by using a weight vector $\boldsymbol{w}$ that is normal to the hyperplane. The weight vector projects the hyperplane to define the decision boundary ($\boldsymbol{c}\,\boldsymbol{w}$) (striped line) that separates the two classes. Data points are likewise projected along $\boldsymbol{w}$. Left: the hyperplane follows the coordinate system. Right: The hyperplane is not aligned with the coordinate system and has a rotation angle.

In summary, to classify if a point $\boldsymbol{x}$ belongs to class 2 using Eq. 4 we multiply it with $\boldsymbol{w}$ which projects the point into the same orientation as the hyperplane for maximal class separation and then we subtract the bias. If $\boldsymbol{y}_{\boldsymbol{C}\in\boldsymbol{2}}(\boldsymbol{x}) > \log(T)$ we have a score defining that the projected data point is closer to class 2 than class 1 and is classified as class 2 (Note, $\log(T) = 0$ in Eq 3). Actually, Eq. 4 only states a score if a data point is belonging to class C2 i.e. $\boldsymbol{y}_{\boldsymbol{C}\in\boldsymbol{2}}(\boldsymbol{x})$, but we can inverse Eq. 2 to define a model for belonging to C1 i.e., $y_{C\in 1}(x)$ by defining a weight vector and bias for class 1.
When knowing the weight vectors and biases for each class we can from the scoring in Eq. 4 calculate the class probability of a point belonging to a class. Since y(x) are log-values we take the exp(y(x)) for a given class and divide it by the marginal probability P(x) i.e., the denominator in Eq. 1. The P(x) is calculated for each data point as the sum of class probabilities. Example, for the first data point $\boldsymbol{x}_1$: $P(\boldsymbol{x}_1) = \sum_{i=1}^{c} P(\boldsymbol{x}_1|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_0,)\boldsymbol{P}(\boldsymbol{C}_i)$.

**The Fisher's linear discriminant classifier**

The LDA classifier model assumptions are not correct if the variances of the two gaussian distributions are not equal or isotropic. In this case, the hyperplane of the LDA is not guaranteed to optimally separate the two classes as illustrated in Figure 2A. However, if changing the model assumptions to account for non-equal and non-isotropic class variances, the projections of the hyperplane, and decision boundary will ensure a better class separation as shown in Figure 2B. This will result in more precise classifications as shown in Figure 2(Right vs Left).

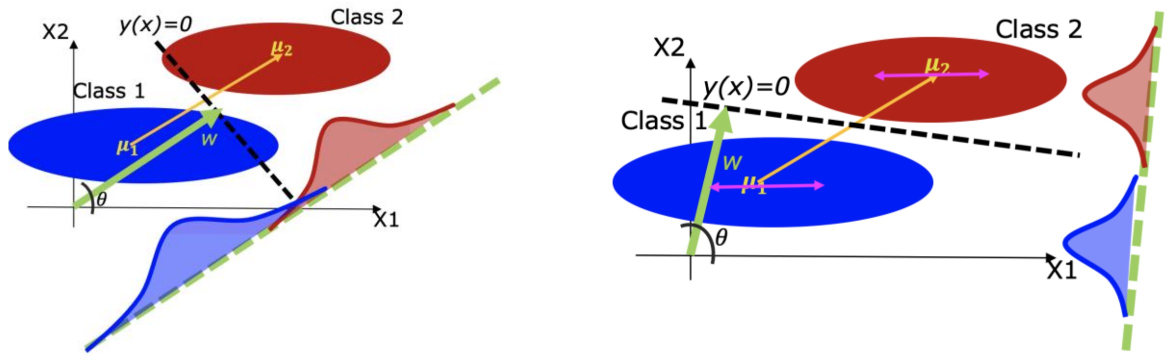

Figure 2 Illustration that the two class distributions have different covariances and are non-isotropic not assumed by the LDA model that risk fails to place the hyperplane for optimal class separation (Left). Right: If we make a new model i.e., Fisher's LDA assuming that covariances can be different and non-isotropic the hyperplane will be placed for optimal class separation.

Fisher's discriminate classifier assumptions account for non-equal and non-isotropic class variances by taking the ratio of *between-class covariance* and the *total within covariance*. The weight vector is to project the hyperplane, so it minimizes the ratio between the two. For this, we can express a cost function to find $\boldsymbol{w}$. The between-class covariance is $S_B = (\boldsymbol{\mu_2} - \boldsymbol{\mu_1})^T(\boldsymbol{\mu_2} - \boldsymbol{\mu_1})$ and the total within-covariance is $S_w = \boldsymbol{\Sigma_1} + \boldsymbol{\Sigma_2}.$ Now we search for a solution for a weight vector and bias that minimizes the ratio between the two covariances.

$$\boldsymbol{J}(\boldsymbol{w}) = \frac{\boldsymbol{w^T S_B w}}{\boldsymbol{w^T S_w w}}$$

This defines the cost function that we differentiate and set to zero to find an optimal solution for $\boldsymbol{w}$:

$$\frac{\boldsymbol{\partial j(w)}}{\boldsymbol{w}} = \boldsymbol{0}$$

The weight vector, $\boldsymbol{w}$, now based on Fisher's discriminate classifier model expresses $w = S_w^{-1}(\boldsymbol{\mu_2} - \boldsymbol{\mu_1})$ and is used in Eq 4. The bias, $\boldsymbol{c}$, is the same as for the LDA as we still wish the decision boundary to be placed halfway between the two class means. Note, the Fisher's model assumptions allow both equal and unequal class covariances, as well as isotropic and non-isotropic covariances, hence extending the model capabilities compared with the LDA.
When knowing the expression of $\boldsymbol{w}$ and $\boldsymbol{wo}$ we can use Fisher's linear discriminant classifier to classifier if a sample belongs to a class or not as well as to estimate the class probability in the same way as described for the LDA.

# Exercise 6b - Advanced segmentation. Fisherman's Linear discriminant analysis for segmentation

## Introduction

The exercise is to the extent of the pixel-wise classification problem from being based on the intensity histogram of a single image modality to combining multiple image modalities. Hence, here we wish to segment image features into two classes by training a classifier based on the intensity information from multiple image modalities. Multiple-image modalities just mean a series of images that contain different but complementary image information of the same object. It is assumed that the image modalities have the same size, so we have a pixel-to-pixel correspondence between the two images. An image feature is an identifiable object in the image e.g., of a dog, moon rocket, or brain tissue types that we wish to segment into individual classes.

Here we aim to segment two types of brain tissues into two feature classes. To improve the segmentation, we wish to combine two MRI image modalities instead of a single: one is a "T1 weighted MRI" and the other is a "T2 weighted MRI". Both are acquired at the same time.

Exercise - You simply go step-by-step and fill in the command lines and answer/discuss the questions (Q1-Q12).

## Theory

### The Linear Discriminate Classifier

As a classifier, we will use a class of linear discriminate functions that aims to place a hyperplane in the multi-dimensional feature space that acts as a decision boundary to segment two features into classes. Since we only look at image intensities of two image modalities our multi-dimensional feature space is a 2D intensity histogram. The linear discriminant classifier is based on the Bayesian theory where the posterior probability is the probability of voxel x belonging to class $C_i$. The voxel x belongs to the class with the highest posterior probability.

You can find an **important** description of the theory behind LDA in - Exercise theory

## Learning Objectives

1. Implement, train and evaluate multi-dimensional segmentation using a Linear Discriminate classifier i.e. Fisherman' Linear discriminant analysis
2. To visualise the 1D intensity histograms of two different image modalities that contain different intensity information of the same image features.

3. To identify the expected intensity thresholds in each of the 1D histograms that best segment the same feature in the two image modalities.
4. To visually the 2D histogram of two image modalities that map the same object but with different intensity information.
5. To interpret the 2D histogram information by identifying clusters of 2D intensity distributions and relate these to features in the images.
6. To draw an expected linear hyper plane in the 2D histogram that best segment and feature in the two image modalities
7. To extract training data sets and their corresponding class labels from expert drawn regions-of-interest data, and map their corresponding 2D histogram for visual inspection
8. To relate the Bayesian theory to a linear discriminate analysis classifier for estimating class probabilities of segmented features.
9. To judge if the estimated linear or a non-linear hyper plane is optimal placed for robust segmentation of two classes.

### Installing Python packages

In this exercise, we will be using numpy, scipy and scikit-image. You should have these 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:

- Exercise data

ex6_ImgData2Load.mat contains all image and ROI data which are loaded into the variables:

- **ImgT1** One axial slice of brain using T1W MRI acquisition
- **ImgT2** One axial slice of brain using T2W MRI acquisition
- **ROI_WM** Binary training data for class 1: Expert identification of voxels belonging to tissue type: White Matter
- **ROI_GM** Binary training data for class 2: Expert identification of voxels belonging to tissue type: Grey Matter

LDA.py A python function that realise the Fisher's linear discriminant analyse as described in Note for the lecture.

## Image Segmentation

Start by importing some useful functions:

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio
```

And the data:

```python
in_dir = 'data/'
in_file = 'ex6_ImagData2Load.mat'
data = sio.loadmat(in_dir + in_file)
ImgT1 = data['ImgT1']
ImgT2 = data['ImgT2']
ROI_GM = data['ROI_GM'].astype(bool)
ROI_WM = data['ROI_WM'].astype(bool)
```

## Exercise 1

Display both the T1 and T2 images, their 1 and 2D histograms and scatter plots. Tips: Use the `plt.imshow()`, `plt.hist()`, `plt.hist2d()` and `plt.scatter()` functions Add relevant title and label for each axis. One can use `plt.subplots()` to show more subfigures in the same figure. Remove intensities from background voxels for 1D and 2D histograms.

imshow image coordinates

The two MRI image modalities contain different types of intensity classes:

1. (Orange): The White Matter (WM) is the tissue type that contain the brain network - like the cables in the internet. The WM ensure the communication flow between functional brain regions.
2. (Yellow): The Grey Matter (GM) is the tissue type that contain the cell bodies at the end of the brain network and are the functional units in the brain. The functional units are like CPUs in the computer. They are processing our sensorial input and are determining a reacting to these. It could be to start running.
3. (Magenta): Cerebrospinal fluid (CSF) which is the water in the brain
4. (Green): Background of the image

**Q1**: What is the intensity threshold that can separate the GM and WM classes (roughly) from the 1D histograms?

**Q2**: Can the GM and WM intensity classes be observed in the 2D histogram and scatter plot?

## Exercise 2

Place trainings examples i.e. ROI_WM and ROI_GM into variables C1 and C2 representing class 1 and class 2 respectively. Show in a figure the manually expert drawings of the C1 and C2 training examples.

*Tips*: use `plt.imshow()`

**Q3**: Does the ROI drawings look like what you expect from an expert?

## Exercise 3

For each binary training ROI find the corresponding training examples in ImgT1 and ImgT2. Later these will be extracted for LDA training.

*Tips*: If you are a MATLAB-like programming lover, you may use the `np.argwhere()` function appropriately to return the index to voxels in the image full filling e.g. intensity values >0 hence belong to a given class. Name the index variables qC1 and qC2, respectively.

**Q4**: What is the difference between the 1D histogram of the training examples and the 1D histogram of the whole image? Is the difference expected?

## Exercise 4

Make a training data vector (X) and target class vector (T) as input for the `LDA()` function. T and X should have the same length of data points.

**X**: Training data vector should first include all data points for class 1 and then the data points for class 2. Data points are the two input features ImgT1, ImgT2

**T**: Target class identifier for X where '0' are Class 1 and a '1' is Class 2.

*Tip: Read the documentation of the provided LDA function to understand the expected input dimensions.*

## Exercise 5

Make a scatter plot of the training points of the two input features for class 1 and class 2 as green and black circles, respectively. Add relevant title and labels to axis

**Q5**: How does the class separation appear in the 2D scatter plot compared with 1D histogram. Is it better?

## Exercise 6

Train the linear discriminant classifier using the Fisher discriminant function and estimate the weight-vector coefficient W (i.e. $w_0$ and $w$) for classification given X and T by using the `W=LDA()` function. The LDA function outputs W=[[w01, w1]; [w02, w2]] for class 1 and 2 respectively.

*Tip: Read the Bishop note on Chapter 4.*

```
W = LDA(X,T)
```

## Exercise 7

Apply the linear discriminant classifier i.e. perform multi-modal classification using the trained weight-vector $W$ for each class: It calculates the linear score

$Y$ for **all** image data points within the brain slice i.e. $y(x) = w + w_0$. Actually, $y(x)$ is the $\log(P(Ci|x))$.

```
Xall= np.c_[ImgT1.ravel(), ImgT2.ravel()]
Y = np.c_[np.ones((len(Xall), 1)), Xall] @ W.T
```

## Exercise 8

Perform multi-modal classification: Calculate the posterior probability i.e. $P(C_1|X)$ of a data point belonging to class 1

*Note: Using Bayes [Eq 1]: Since $y(x)$ is the log of the posterior probability [Eq2] we take $\exp(y(x))$ to get $P(C_1|X) = P(X|\mu, \sigma)P(C_1)$ and divide with the marginal probability $P(X)$ as normalisation factor.*

```
PosteriorProb = np.clip(np.exp(Y) / np.sum(np.exp(Y),1)[:, np.newaxis]), 0, 1)
```

## Exercise 9

Apply segmentation: Find all voxles in the T1w and T2w image with $P(C_1|X) > 0.5$ as belonging to Class 1. You may use the `np.where()` function. Similarly, find all voxels belonging to class 2.

## Exercise 10

Show scatter plot of segmentation results as in 5.

**Q6** Can you identify where the hyperplane is placed i.e. y(x)=0?

**Q7** Is the linear hyper plane positioned as you expected or would a non-linear hyper plane perform better?

**Q8** Would segmentation be as good as using a single image modality using thresholding?

**Q9** From the scatter plot does the segmentation results make sense? Are the two tissue types segmented correctly.

## Exercise 11

**Q10** Are the training examples representative for the segmentation results? Are you surprised that so few training examples perform so well? Do you need to be an anatomical expert to draw these?

**Q11** Compare the segmentation results with the original image. Are the segmentation results satisfactory? Why not?

**Q12** Is one class completely wrong segmented? What is the problem?

# Exercise 7 - Geometric transformations and landmark based registration

In this exercise, we will explore geometric transformations of images and landmark based registration.

## Learning Objectives

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

1. Use `skimage.transform.rotate` to rotate an image using different rotation centers, different background filling strategies (constant, reflection, warping) and automatic scaling of the output image.
2. Construct an Euclidean (translation plus rotation) transform using `skimage.transform.EuclideanTransform`.
3. Apply a given transform to an image using `skimage.transform.warp`.
4. Compute and apply the inverse of a transform.
5. Construct a similarity (translation, rotation plus scale) transform using `skimage.transform.SimilarityTransform`.
6. Use the `skimage.transform.swirl`to transform images.
7. Compute and visualize the blend of two images.
8. Manually place landmarks on an image.
9. Visualize sets of landmarks on images.
10. Compute the objective function $F$ between two sets of landmarks.
11. Use the `estimate` function to estimate the optimal transformation between two sets of landmarks.
12. Use the `skimage.transform.matrix_transform` to transform a set of landmarks.
13. Implement and test a program that can transform and visualize images from a video stream.

## Installing Python packages

In this exercise, we will be using both scikit-image and OpenCV. You should have these 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/tree/main/exercises/Ex7-GeometricTransformationsAndRegistration/data)

## Geometric transformations on images

The first topic is how to apply geometric transformations on images.

Let us start by defining a utility function, that can show two images side-by-side:

```
def show_comparison(original, transformed, transformed_name):
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4), sharex=True,
                                   sharey=True)
    ax1.imshow(original)
    ax1.set_title('Original')
    ax1.axis('off')
    ax2.imshow(transformed)
    ax2.set_title(transformed_name)
    ax2.axis('off')
    io.show()
```

also import some useful functions:

```
import matplotlib.pyplot as plt
import math
from skimage.transform import rotate
from skimage.transform import EuclideanTransform
from skimage.transform import SimilarityTransform
from skimage.transform import warp
from skimage.transform import swirl
```

## Image rotation

One of the most useful and simple geometric transformation is rotation, where an image is rotated around a point.

We start by some experiments on the image called **NusaPenida.png**. It can be found in the exercise material

### Exercise 1

Read the **NusaPenida.png** image and call it **im_org**. It can be rotated by:

```
# angle in degrees - counter clockwise
rotation_angle = 10
rotated_img = rotate(im_org, rotation_angle)
show_comparison(im_org, rotated_img, "Rotated image")
```

Notice, that in this function, the angle should be given in degrees.

By default, the image is rotated around the center of the image. This can be changed by manually specifying the point that the image should be rotated around (here (0, 0)):

```
rot_center = [0, 0]
rotated_img = rotate(im_org, rotation_angle, center=rot_center)
```

### Exercise 2

Experiment with different center points and notice the results.

As seen, there are areas of the rotated image that is filled with a background value. It can be controlled how this background filling shall behave.

Here the background filling mode is set to **reflect**

```
rotated_img = rotate(im_org, rotation_angle, mode="reflect")
```

### Exercise 3

Try the rotation with background filling mode **reflect** and **wrap** and notice the results and differences.

It is also possible to define a constant fill value. Currently, sci-kit image only supports a single value (not RGB).

### Exercise 4

Try to use:

```
rotated_img = rotate(im_org, rotation_angle, resize=True, mode="constant", cval=1)
```

with different values of `cval` and notice the outcomes.

By default, the rotated output image has the same size as the input image and therefore some parts of the rotated image are cropped away. It is possible to automatically adjust the output size, so the rotated image fits into the resized image.

### Exercise 5

Test the use of automatic resizing:

```
rotated_img = rotate(im_org, rotation_angle, resize=True)
```

also combine resizing with different background filling modes.

## Euclidean image transformation

An alternative way of doing geometric image transformations is to first construct the transformation and then apply it to the image. We will start by the **Euclidean** image transformation that consists of a rotation and a translation. It is also called a *rigid body transformation*.

**Exercise 6**

Start by defining the transformation:

```
# angle in radians - counter clockwise
rotation_angle = 10.0 * math.pi / 180.
trans = [10, 20]
tform = EuclideanTransform(rotation=rotation_angle, translation=trans)
print(tform.params)
```

it can be seen in the print statement that the transformation consists of a *3 x 3 matrix*. The matrix is used to transform points using **homogenous coordinates**. Notice that the angle is defined in radians in this function.

**Exercise 7**

The computed transform can be applied to an image using the `warp` function:

```
transformed_img = warp(im_org, tform)
```

Try it.

**Note:** The `warp` function actually does an *inverse* transformation of the image, since it uses the transform to find the pixels values in the input image that should be placed in the output image.

## Inverse transformation

It is possible to get the inverse of a computed transform by using `tform.inverse`. An image can then be transformed using the invers transform by:

```
transformed_img = warp(im_org, tform.inverse)
```

**Exercise 8**

Construct a Euclidean transformation with only rotation. Test the transformation and the invers transformation and notice the effect.

## Similarity transform of image

The `SimilarityTransform` computes a transformation consisting of a translation, rotation and a scaling.

**Exercise 9**

Define a `SimilarityTransform` with an angle of $15^o$, a translation of (40, 30) and a scaling of 0.6 and test it on the image.

### The swirl image transformation

The **swirl** image transform is a non-linear transform that can create interesting visual results on images.

#### Exercise 10

Try the swirl transformation:

```
str = 10
rad = 300
swirl_img = swirl(im_org, strength=str, radius=rad)
```

it is also possible to change the center of the swirl:

```
str = 10
rad = 300
c = [500, 400]
swirl_img = swirl(im_org, strength=str, radius=rad, center=c)
```

try with different centers and notice the results.

## Landmark based registration

The goal of landmark based registration is to align two images using a set of landmarks placed in both images. The landmarks need to have *correspondence* meaning that the landmarks should be placed on the same anatomical spot in the two images.

There are two photos of hands: **Hand1.jpg** and **Hand2.jpg** and the goal is to transform **Hand1** so it fits on top of **Hand2**. In this exercise we call Hand1 one for the *source* (src) and Hand2 for the *destination* (dst).

#### Exercise 11

Start by reading the two images into *src_img* and *dst_img*. Visualize their overlap by:

```
blend = 0.5 * img_as_float(src_img) + 0.5 * img_as_float(dst_img)
io.imshow(blend)
io.show()
```

### Manual landmark annotation

We will manually placed landmarks on the two images to align the them.

#### Exercise 12

We have manually placed a set of landmarks on the source image. They can be visualized by:

```
src = np.array([[588, 274], [328, 179], [134, 398], [260, 525], [613, 448]])

plt.imshow(src_img)
plt.plot(src[:, 0], src[:, 1], '.r', markersize=12)
plt.show()
```

**Exercise 13**

You should now place the same landmarks on the destination image.

In imshow you can see the pixel coordinates of the cursor:

imshow image coordinates

Use this to find the coordinates of the sought landmarks and put them into a `dst` variable.

Plot the landmarks to verify they are correct:

```
fig, ax = plt.subplots()
ax.plot(src[:, 0], src[:, 1], '-r', markersize=12, label="Source")
ax.plot(dst[:, 0], dst[:, 1], '-g', markersize=12, label="Destination")
ax.invert_yaxis()
ax.legend()
ax.set_title("Landmarks before alignment")
plt.show()
```

To calculate how well two sets of landmarks are aligned, we can compute the *objective function*:

$$F = \sum_{i=1}^{N} \|a_i - b_i\|^2 \ ,$$

here $a_i$ are the landmarks in the destination image and $b_i$ are the landmarks in the source image.

**Exercise 14**

Compute $F$ from your landmarks. It can for example be done like:

```
e_x = src[:, 0] - dst[:, 0]
error_x = np.dot(e_x, e_x)
e_y = src[:, 1] - dst[:, 1]
error_y = np.dot(e_y, e_y)
f = error_x + error_y
print(f"Landmark alignment error F: {f}")
```

The optimal Euclidean transformation that brings the source landmarks over in the destination landmarks can be found by:

```
tform = EuclideanTransform()
tform.estimate(src, dst)
```

The found transform can be applied to the source points by:

```
src_transform = matrix_transform(src, tform.params)
```

**Exercise 15**

Visualize the transformed source landmarks together with the destination landmarks. Also compute the objective function $F$ using the transformed points. What do you observe?

**Exercise 16**

We can now apply the transformation to the source image. Notice that we use the inverse transform due to the inverse mapping in the image resampling:

```
warped = warp(src_img, tform.inverse)
```

Show the warped image and also try to blend the warped image destination image like in exercise 11. What do you observe?

## Video transformations

Now try to make a small program, that acquires video from your webcam/telephone, transforms it and shows the output. In the exercise material there is a program that can be modified.

By default, the program acquires a colour image and rotates it. There is a counter that is increased every frame and that counter can be used to modify the transformation (for example the rotation angle). The program also measures how many milliseconds the image processing takes.

**Exercise 16**

Run the example program and notice how the output image rotates.

**Exercise 17**

Modify the program so it performs the **swirl** transform on the image. The parameters of the swirl transform can be changed using the counter. For example:

```
str = math.sin(counter / 10) * 10
```

Try this and also try to change the other transform parameters using the counter.

## References

- sci-kit image transformations
- sci-kit image rotation

- transformation example
- swirl transform

# Exercise 8 - Cats, Cats, and EigenCats

Are you sad that you have watched all cat movies and seen all cat photos on the internet? Then be sad no more - in this exercise we will make a *Cat Synthesizer* where you can create all the cat photos you will ever need!

Secondly, a very unfortunate event happened so you are now in a situation, where you need to find a *perfect new twin cat*.

To be able to do these wonderful things we will harness the power of image based *principal component analysis*. The methods we will use, can be called *classical machine learning*.

Missing Cat

## Learning Objectives

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

1. Preprocess a batch of images so they can be used to machine learning. Preprocessing can include geometric transformations, intensity transformations and image cropping.
2. Use the python function `glob` to find all files with a given pattern in a folder.
3. Create an empty data matrix that can hold a given set of images and a given number of measurements per image.
4. Compute the number of features per image using the height, width and the number of channels in the image.
5. Use the function `flatten` to convert an image into a 1-D vector.
6. Create an image from a 1-D vector by using the `reshape` function.
7. Create an unsigned byte image from a float image using pixel value scaling and pixel type conversion.
8. Read a set of images and put their pixel values into a data matrix.
9. Compute an average image using the data matrix.
10. Visualize an average image
11. Preprocess one image so it can be used in machine learning.
12. Use sum-of-squared pixel differences (SSD) to compare one image with all images in a training set.
13. Identify and visualize the images in the training set with the smallest and largest SSD compared to a given image.
14. Do a principal component analysis (PCA) of the data matrix using the sci-kit learn PCA
15. Select the number of components that should be computed in the PCA.
16. Extract and visualize the amount of the total variation that each principal component explains.
17. Project the data matrix into PCA space.
18. Plot the first two dimensions of the PCA space. The PCA space is the positions of each sample when projected onto the principal components.

19. Identify and visualize the images that have extreme positions in PCA. For example the images that have the largest and smallest coordinates in PCA space.
20. Compute and visualize a synthetic image by adding linear combinations of principal components to the average image.
21. Select suitable weights based on the PCA space for synthesizing images.
22. Compute and visualize the major modes of variation in the training set, by sampling along the principal components.
23. Generate random synthetic images that lies within the PCA space of the training set.
24. Project a given image into PCA space
25. Generate a synthetich version of an image by using the image position in PCA space.
26. Compute the Euclidean distance in PCA space between a given image and all other images.
27. Identify and visualize the images in the training set with the smallest and largest Euclidean distance in PCA space to a given image.
28. Use `argpartition` to find the N closest images in PCA space to a given image.

## Importing required Python packages

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

Let us start with some imports:

```
from skimage import io
from skimage.util import img_as_ubyte
import matplotlib.pyplot as plt
import numpy as np
import glob
from sklearn.decomposition import PCA
from skimage.transform import SimilarityTransform
from skimage.transform import warp
import os
import pathlib
```

## Exercise data and material

The data and material needed for this exercise can be found here: exercise data and material

The main part of the data is a large database of photos of cats, where there are also a set of landmarks per photo. You should download the data here.

**IMPORTANT:** You can start by working with the smaller data set with 100 cats found here: smaller photo database of 100 cats. Later you can use the full data set and see if your computer has enough RAM to handle it.

Start by unpacking all the training photos in folder you choose. For example `training_data`.

## Preprocessing data for machine learning

The photos contains cats in many situations and backgrounds. To make it easier to do machine learning, we will *preprocess* the data, so the photos only contains the face of the cat. Preprocessing is and important step in most machine learning approaches.

The preprocessing steps are:

- Define a model cat (`ModelCat.jpg`) with associated landmarks (`ModelCat.jpg.cat`)
- For each cat in the training data:
  - Use landmark based registration with a *similarity transform* to register the photo to the model cat
  - Crop the registered photo
  - Save the result in a fold called **preprocessed**

**Exercise 1:** *Preprocess all image in the training set. To do the preprocessing, you can use the code snippets supplied* here. There is also a **Model Cat** supplied.

The result of the preprocessing is a directory containing smaller photos containing cat faces. All the preprocessed photos also have the same size.

## Gathering data into a data matrix

To start, we want to collect all the image data into a data matrix. The matrix should have dimensions `[n_samples, n_features]` where **n_samples** is the number of photos in our training set and **n_features** is the number of values per image. Since we are working with RGB images, the number of features are given by `n_features = height * width * channels`, where `channels = 3`.

The data matrix can be constructed by:

- Find the number of image files in the **preprocessed** folder using `glob`. Look at the `preprocess_all_cats` function to get an idea of how to use `glob`.
- Read the first photo and use that to find the height and width of the photos
- Set **n_samples** and **n_features**
- Make an empty matrix `data_matrix = np.zeros((n_samples, n_features))`
- Read the image files one by one and use `flatten()` to make each image into a 1-D vector (flat_img).
- Put the image vector (flat_img) into the data matrix by for example `data_matrix[idx, :] = flat_img` , where idx is the index of the current image.

**Exercise 2:** *Compute the data matrix.*

## Compute and visualize a mean cat

In the data matrix, one row is one cat. You can therefore compute an average cat, **The Mean Cat** by computing one row, where each element is the average of the column.

**Exercise 3:** *Compute the average cat.*

You can use the supplied function `create_u_byte_image_from_vector` to create an image from a 1-D image vector.

**Exercise 4:** *Visualize the Mean Cat*

## Find a missing cat or a cat that looks like it (using image comparison)

You have promised to take care of your neighbours cat while they are on vacation. But...Oh! no! You were in such a hurry to get to DTU that you forgot to close a window. Now the cat is gone!!! What to do?

**Exercise 5:** *Decide that you quickly buy a new cat that looks very much like the missing cat - so nobody notices*

Luckily, the training set is actually photos of cats that are in a *get a new cat cheap* nearby store.

To find a cat that looks like the missing cat, you start by comparing the missing cat pixels to the pixels of the cats in the training set. The comparison between the missing cat data and the training data can be done using the sum-of-squared differences (SSD).

**Exercise 6:** *Use the `preprocess_one_cat` function to preprocess the photo of the poor missing cat*

**Exercise 7:** *Flatten the pixel values of the missing cat so it becomes a vector of values.*

**Exercise 8:** *Subtract you missing cat data from all the rows in the data_matrix and for each row compute the sum of squared differences. This can for example be done by:*

```
sub_data = data_matrix - im_miss_flat
sub_distances = np.linalg.norm(sub_data, axis=1)
```

**Exercise 9:** *Find the cat that looks most like your missing cat by finding the cat, where the SSD is smallest. You can for example use `np.argmin`.*

**Exercise 10:** *Extract the found cat from the data_matrix and use `create_u_byte_image_from_vector` to create an image that can be visu-*

*alized. Did you find a good replacement cat? Do you think your neighbour will notice? Even with their glasses on?*

**Exercise 11:** *You can use `np.argmax` to find the cat that looks the least like the missing cat.*

You can also use your own photo of a cat (perhaps even your own cat). To do that you should:

- Place a jpg version of your cat photo in the folder where you had your missing cat photo. Call it for example **mycat.jpg**
- Create a landmark file called something like **mycat.jpg.cat**. It is a text file.
- In the landmark file you should create three landmarks: `3 189 98 235 101 213 142` . Here the first `3` just say there are three landmarks. The following 6 numbers are the (x, y) positions of the right eye, the left eye and the nose. You should manually update these numbers.
- Use the `preprocess_one_cat` function to preprocess the photo
- Now you can do the above routine to match your own cat.

**Optional Exercise:** *Use a photo of your own cat to find its twins*

## Principal component analysis on the cats

We now move to more classical machine learning on cats. Namely Principal component analysis (PCA) analysis of the cats image.

To compute the PCA, we use the sci-kit learn PCA. Note that this version of PCA automatically *centers* data. It means that it will subtract the average cat from all cats for you.

**Exercise 12:** *Start by computing the first 50 principal components:*

```
print("Computing PCA")
cats_pca = PCA(n_components=50)
cats_pca.fit(data_matrix)
```

This might take some time. If your compute can not handle so many images, you can manually move or delete som photos out of the **preprocessed** folder before computing the data matrix.

The amount of the total variation that each component explains can be found in `cats_pca.explained_variance_ratio_`.

**Exercise 13:** *Plot the amount of the total variation explained by each component as function of the component number.*

**Exercise 14:** *How much of the total variation is explained by the first component?*

We can now project all out cat images into the PCA space (that is 50 dimensional):

**Exercise 15:** *Project the cat images into PCA space*:

```
components = cats_pca.transform(data_matrix)
```

Now each cat has a position in PCA space. For each cat this position is 50-dimensional vector. Each value in this vector describes *how much of that component* is present in that cat photo.

We can plot the first two dimensions of the PCA space, to see where the cats are placed. The first PCA coordinate for all the cats can be found using `pc_1 = components[:, 0]` .

**Exercise 16:** *Plot the PCA space by plotting all the cats first and second PCA coordinates in a (x, y) plot*

## Cats in space

We would like to explore what the PCA learnt about our cats in the data set.

### Extreme cats

We start by finding out which cats that have the most *extreme coordinates* in PCA space.

**Exercise 17:** *Use `np.argmin` and `np.argmax` to find the ids of the cats that have extreme values in the first and second PCA coordinates. Extract the cats data from the data matrix and use `create_u_byte_image_from_vector` to visualize these cats. Also plot the PCA space where you plot the extreme cats with another marker and color.*

**Exercise 18:** *How do these extreme cat photo look like? Are some actually of such bad quality that they should be removed from the training set? If you remove images from the training set, then you should run the PCA again. Do this until you are satisfied with the quality of the training data.*

### The first synthesized cat

We can use the PCA to make a so-called **generative model** that can create synthetic samples from the learnt data. It is done by adding a linear combination of principal components to the average cat image:

$$I_{\text{synth}} = I_{\text{average}} + w_1 * P_1 + w_2 * P_2 + ... + w_k * P_k \ ,$$

where we $P_1$ is the first principal component, $P_2$ the second and so on. Here we use $k$ principal components.

The principal components are stored in `cats_pca.components_`. So the first principal component is `cats_pca.components_[0, :]` .

**Exercise 19:** *Create your first fake cat using the average image and the first principal component. You should choose experiment with different weight values (w) :*

```
synth_cat = average_cat + w * cats_pca.components_[0, :]
```

**Exercise 20:** *Use `create_u_byte_image_from_vector` visualize your fake cat.*

You can use the PCA plot we did before to select some suitable values for w.

**Exercise 21:** *Synthesize some cats, where you use both the first and second principal components and select their individual weights based on the PCA plot.*

## The major cat variation in the data set

A very useful method to get an overview of the **major modes of variation** in a dataset is to synthesize the samples that are lying on the outer edges of the PCA space.

If we for example move a distance out of the first principal axis we can synthesize the cat image there. In this case we will try to move to $\pm\sqrt{(\text{explained variance})}$, where *explained variance* is the variance explained by that principal component. In code, this will look like:

```
synth_cat_plus = average_cat + 3 * np.sqrt(cats_pca.explained_variance_[m]) * cats_pca.compo
synth_cat_minus = average_cat - 3 * np.sqrt(cats_pca.explained_variance_[m]) * cats_pca.comp
```

here **m** is the principal component that we are investigating.

**Exercise 22:** *Synthesize and visualize cats that demonstrate the first three major modes of variation. Try show the average cat in the middle of a plot, with the negative sample to the left and the positive to the right. Can you recognise some visual patterns in these modes of variation?*

## The Cat Synthesizer (EigenCats)

We are now ready to make true cat synthesizer, where cat images are synthesized based on random locations in PCA space. You can start by setting your `synth_cat = average_cat`. Then you can add all the components you want by for example (this number should be less or equal to the number of components we asked the PCA to compute):

```
n_components_to_use = 10
synth_cat = average_cat
for idx in range(n_components_to_use):
    w = random.uniform(-1, 1) * 3 * np.sqrt(cats_pca.explained_variance_[idx])
    synth_cat = synth_cat + w * cats_pca.components_[idx, :]
```

**Exercise 23:** *Generate as many cat photos as your heart desires..*

## Cat identification in PCA space

Now back to your missing cat. We could find similar cats by computing the difference between the missing cat and all the photos in the databased. Imagine that you only needed to store the 50 weights per cats in your database to do the same type of identification?

**Exercise 24:** *Start by finding the PCA space coordinates of your missing cat:*

```
im_miss = io.imread("data/cats/MissingCatProcessed.jpg")
im_miss_flat = im_miss.flatten()
im_miss_flat = im_miss_flat.reshape(1, -1)
pca_coords = cats_pca.transform(im_miss_flat)
pca_coords = pca_coords.flatten()
```

The `flatten` calls are needed to bring the arrays into the right shapes.

**Exercise 25:** *Plot all the cats in PCA space using the first two dimensions. Plot your missing cat in the same plot, with another color and marker. Is it placed somewhere sensible and does it have close neighbours?*

We can generate the synthetic cat that is the closest to your missing cat, by using the missing cats position in PCA space:

```
n_components_to_use = ?
synth_cat = average_cat
for idx in range(n_components_to_use):
    synth_cat = synth_cat + pca_coords[idx] * cats_pca.components_[idx, :]
```

**Exercise 26:** *Generate synthetic versions of your cat, where you change the n_components_to_use from 1 to for example 50.*

We can compute Euclidean distances in PCA space between your cat and all the other cats by:

```
comp_sub = components - pca_coords
pca_distances = np.linalg.norm(comp_sub, axis=1)
```

**Exercise 27:** *Find the id of the cat that has the smallest and largest distance in PCA space to your missing cat. Visualize these cats. Are they as you expected? Do you think your neighours will notice a difference?*

You can also find the n closest cats by using the `np.argpartition` function.

**Exercise 28:** *Find the ids of and visualize the 5 closest cats in PCA space. Do they look like your cat?*

What we have been doing here is what has been used for face identification and face recognition (*Matthew Turk and Alex Pentland: Eigenfaces for Recognition, 1991*)

In summary, we can now synthesize all the cat photos you will only need and we can help people that are loooking for cats that looks like another cat. On top of that, we can now use methods that are considered state-of-the-art before the step into deep learning.

## References

- Cat data set
- sci-kit learn PCA

# Image Analysis
# Exercise - Advanced registration

## Theory

The exercise is to get experience working with 3D image registration pipelines (fig. 1) and the theory part describes the different steps in the pipeline. The focus is on the considerations to be made when selecting the various algorithm parameters to ensure a robust registration result.

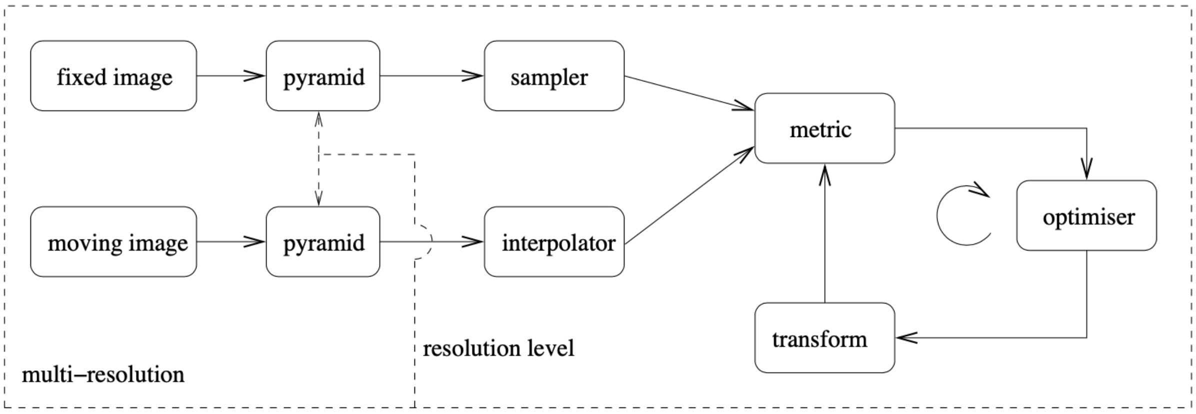

Figure 1: A general outline of an image registration pipeline at its different steps as implemented in the Eleastix toolbox that the exercise and the lecture note are based on (Elastix - The manual by Stefan Klein and Marius Staring).

The exercise will use the "SimpleITK" Python library which is based on the "Advanced registration" lecture note (DTU Learn) on the Elastix toolbox - The manual (chapter 2). The SimpleITK library builds on the algorithms in Elastix but there is no guarantee that the function naming is the same.

In image registration, our aim is to find a geometrical matrix that best registers the moving image with a fixed image of the same scene. Often, the moving image can be named as the "reference" image whereas the fixed image is named as the "template" or the "static" image. In this exercise, we will have different geometrical transformation matrices (the transform step) to be combined and applied to an image or a series of images. Therefore, we wish to combine the transformation matrices in an efficient way - and for this, we will use the Homogeneous coordinate system. We will combine geometrical transformation matrices to an affine matrix and apply it to the moving image by reslicing to place the moving image on the same image grid as the fixed image. In reslicing, we use backward mapping of a data point in the fixed image grid into the moving image and use intensity interpolation between neighboring voxels to find the intensity in the fixed image grid (the interpolation step). There exist different interpolation methods and, in the exercise, we use a simple linear (bilinear) interpolation method. Note that all types of interpolation methods always introduce some degree of blurring and the more advanced interpolation methods e.g., Sinc-interpolation the less blurring is introduced. The price for less blurring is a longer processing time. Further, we will explore the pyramid (the pyramid step) procedure that uses a multi-resolution strategy step to robustly find an optimal affine geometrical transformation matrix (the optimiser and the metric step).

In this exercise, we will work with affine transformations which is a linear geometrical transformation up to 12 degrees of freedom for 3D images i.e., up to 12 free parameters are to be estimated. The 12 parameters are three for the translation, three for rotation, three for scaling, and three for shearing each in the x, y, and z-axis respectively. In this exercise, we will constrain the affine transformation to a ridge transformation including only translation and rotation, reducing the number of free parameters to 6. Hence, the scaling and shearing are set to an identity matrix.

An affine transformation (A) for a 3D image is 3x3 (Eq.1) and is constructed from geometrical transformation matrices being the translation (*$A_T$*), rotation (*$A_R$*), scaling (*$A_S$*), and shearing (*$A_Z$*).

$$A_{T,R,S,Z} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad (1)$$

Often the affine transformation matrix is constituted by a combination of the different geometrical transformations. However, translation is a summation operation i.e., $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = A_T + \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ whereas scaling, rotation, and shearing are matrix multiplications of the input points i.e., $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = A_{S,R,S} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$. Therefore, we wish to find a way to enable multiplication also for the translation matrix so any combination of the geometrical transformations can be combined via matrix multiplication operations.

To combine different matrices via multiplication, we use the Homogeneous coordinate system where we add an extra dimension *W* which is set to 1 (for more information on the Homogeneous coordinate system widely used in computer vision read this blog[1]). Hence, the geometrical matrix for a 3D image becomes a 4D matrix with the form

$$A = \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2)$$

The idea of using the Homogeneous coordinate system is that the translation matrix now can be formulated as

$$A_T = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (3)$$

and the rotation matrix A$_R$ includes three rotations matrices (R$_X$, R$_y$, R$_Z$) that expresses

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_y = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_z = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (4)$$

The scaling (A$_S$) and the shearing (A$_z$) matrices become

$$A_s = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad A_z = \begin{bmatrix} 1 & Sxy & Sxz & 0 \\ Sxy & 1 & Syz & 0 \\ Sxz & Syz & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (5)$$

[1] www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/

Now we can write the 4x4 affine matrix as a multiplication between any of the geometrical transformations for 3D images. Often the standard order of combining the different geometrical transformations is

$$A = A_T * A_R * A_s * A_z \quad (6)$$

In the same way as combining the different geometrical transformations, we can combine a series of affine matrices between images as multiplications. Here, the order in which matrices are combined is important too. Beware, that different image registration programs may use different combinatorial orders, and especially the rotations can be counterclockwise or clockwise. Obviously, that will generate different affine matrices, and care should be taken when combining geometrical matrices obtained from different programs - Therefore, always visually inspect your registration results.

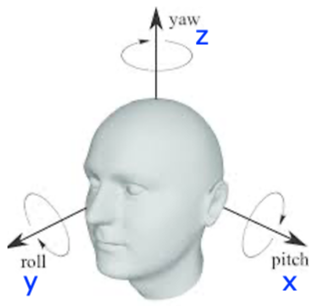

Figure 1: 3D rotation coordinate system

The rotation matrix in Eq 6 defines a 3D rotation coordinate system (Fig. 1) composed of three elements rotations $\alpha, \beta$ *and* $\gamma$ for the $R_X$, $R_y$, and $R_Z$ matrices, respectively (Eq 4). The rotation around the x-axis is named the pitch, the roll is around the y-axis, and the yaw is around the z-axis. The three rotations angles are combined using the Euler angle conventions in the following order

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R_x \cdot R_y \cdot R_z \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

When performing rotation by applying a rotation matrix, it is important to define the center of rotation used also when combining different affine matrices e.g., from a sequence of affine image rotations. The center of rotation is defined as an origin and often uses as default the middle point of the image matrix as is the case in the exercise or at position (0,0) (x,y), or the origin can be user-defined so check it for the implementation you use.

**Finding the optimal parameters for an affine registration matrix**

To find the optimal parameter set for a geometrical transformation that optimally registers the fixed and the moving image we use a similarity matrix as a cost function, and a numerical optimization algorithm (the optimiser step, the metric step, and the transform step). In the exercise, we use the mean square error as a similarity metric (Eq 2.5 in the lecture note). It is fast to estimate and relatively robustly can find the global minimal hence the optimal parameter set. The best robustness is if the structural intensity information in the fixed and the moving image is the same. If that is not the case, the mean square error metric may have more local minima leading to a higher risk of a sub-optimal registration solution. Alternatively, the mutual information metric is based on image histogram matching and is especially good if the fixed and moving images are having similar structural information but contain different intensity information i.e., different intensity histograms. The "price" to pay for the more robust similarity metric is a longer processing time.

The optimizer algorithm iteratively searches in small steps the parameter space of the cost function to find a global minimum. At this point, we expect to find the solution for an optimal parameter set of the affine matrix that best registers the moving image to the fixed image. There exist many different optimizer algorithms and we select the Powell optimizer that better defines the gradient direction in

each iteration than the gradient decent algorithm described in the lecture. Still, one needs to select the step length along the gradient direction and the maximal number of iterations. If the step length is too small one becomes more sensitive to local minima whereas too big a step length can pass the global minimum. If too few iterations are selected, the global minimum will not be reached whereas for too many steps the processing can take a long time especially if no clear solution exists e.g., in the case of very noisy images. So, a suitable number of iterations and not too large nor too small step lengths are to be selected - and good guidance is using the default in the program or simply finding it empirically.
Some types of images can have a risk of containing many local minima i.e., several "local" structures look like others, and we have many suboptimal solutions to the registration problem. This is often the case in MRI where the finer anatomical structures such as cortical folding look the same in many places in the brain. It could also be the case for noisy images.
The pyramidal multi-resolution approach (the pyramid step) is a way to reduce the number of local minima. As the name indicates it works in different image resolution levels like a pyramid shape. The procedure is as the following. We start at the lowest image resolution. Here, one only sees the major/coarse brain structures outlined in the moving and fixed images. For this resolution level, a first rough estimate of the transformation matrix is found. Next at the finer resolution, we use as a starting guess the transformation matrix obtained from the previous lower resolution step and refine it based on the finer anatomical details appearing at the finer resolution. The procedure is typically repeated at three levels i.e., factors of four, two, and one times the image resolution. Instead of changing the image resolution, one can use different sizes of Gaussian kernels to introduce blurring which has the same effect. In the exercise, we will use the pyramid step, and one must set the number of levels in the pyramid. Further, for each resolution level, one set the down-sampling factor, the number of iterations for the optimiser, and the size of the Gaussian blurring kernel.

When we finally have found the optimal affine matrix for the moving image it would be nice to be able to inspect the individual transformations i.e., translation, rotation, scaling, and shearing parameters. This is not possible by inspecting the affine matrix outputted by the registration pipeline where the different transformations are multiplied. Therefore, we must decompose the affine matrix back into the different geometrical transformation matrices using a second numerical optimizer algorithm. The Python library we use offers such decomposing possibility of the affine matrix so we easily can evaluate if the transformation parameters are as expected.

**The world-coordinates system versus the grid system**
In most registration programs one just gives as input the fixed and the moving image and asks to find the optimal geometrical transformation matrix. This is most often the case for 2D image registration methods. However, in 3D registration and especially in medical image registrations an extra set of affine matrices are used for both the moving and the fixed image. In the exercise, we ignore these matrices by setting them to an identity matrix. The idea of the two transformation matrices is to convert between two coordinate systems being the word-coordinate and the image data coordinate system. The world-coordinates system describes a reference system of the image in millimeters how the MR image of a person's head is positioned in relation to the coordinate system of the MR scanner. The image coordinate system expresses data indices of the MR image matrix - also references as indices to a data grid. In image registration, the affine matrix describing the world-coordinate system can be

$$A_{world} = \begin{bmatrix} V_x & 0 & 0 & O_x \\ 0 & V_y & 0 & O_y \\ 0 & 0 & V_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

where the diagonal ($V_{x\text{-}z}$) is the same as the scale transformation matrix in Eq (5) but describes the voxel size in millimeters, and $O_{x\text{-}z}$ is the origin of the image. Off-diagonal values are often non-zero if the head is tilted in relation to the coordinate system of the MRI scanner. Naturally, there exists an affine world-coordinate matrix for each image i.e., one for the moving and one for the fixed.
The affine matrix describing the image data grid is simply the geometrical transformation between the moving and the fixed image that we are looking for (Eq 6). The affine transformation for the world-coordinate system of a 3D image is often stored in the header information of the image that most registration programs automatically read which includes the program we are using in the exercise.

# Exercise 9 - Advanced 3D registration

In this exercise, we will use the SimpleITK library to perform 3D image registration. You will familiarize yourself with the registration process, its challenges and the different elements you can tune to improve the registration results.

## Learning Objectives

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

1. Use SimpleITK (https://simpleitk.readthedocs.io/en/master/) for 3D registration.

## Theory

You can find an **important** description of the theory in - Exercise theory (theory/Exercise9_AdvancedImageRegistration_2023.pdf)

## Installing Python packages

In this exercise, we will introduce SimpleITK. SimpleITK is an open-source image analysis toolkit designed to provide a simple and efficient way to access and manipulate 3D image data. It is a simplified, user-friendly interface to the Insight Segmentation and Registration Toolkit (ITK), a widely used image analysis library for image processing. SimpleITK is written in C++, but provides bindings for several programming languages, including Python. You can find more information about SimpleITK here (https://simpleitk.readthedocs.io/en/master/).

You can install SimpleITK with the command `pip install SimpleITK`.

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

# Image Registration

Start by importing some useful functions:

```
import numpy as np
import matplotlib.pyplot as plt
import SimpleITK as sitk
from IPython.display import clear_output
from skimage.util import img_as_ubyte
```

and defining some useful functions:

```
def imshow_orthogonal_view(sitkImage, origin = None, title=None):
    """
    Display the orthogonal views of a 3D volume from the middle of the volume.

    Parameters
    ----------
    sitkImage : SimpleITK image
        Image to display.
    origin : array_like, optional
        Origin of the orthogonal views, represented by a point [x,y,z].
        If None, the middle of the volume is used.
    title : str, optional
        Super title of the figure.

    Note:
    On the axial and coronal views, patient's left is on the right
    On the sagittal view, patient's anterior is on the left
    """
    data = sitk.GetArrayFromImage(sitkImage)

    if origin is None:
        origin = np.array(data.shape) // 2

    fig, axes = plt.subplots(1, 3, figsize=(12, 4))

    data = img_as_ubyte(data/np.max(data))
    axes[0].imshow(data[origin[0], ::-1, ::-1], cmap='gray')
    axes[0].set_title('Axial')

    axes[1].imshow(data[::-1, origin[1], ::-1], cmap='gray')
    axes[1].set_title('Coronal')

    axes[2].imshow(data[::-1, ::-1, origin[2]], cmap='gray')
    axes[2].set_title('Sagittal')

    [ax.set_axis_off() for ax in axes]

    if title is not None:
        fig.suptitle(title, fontsize=16)

def overlay_slices(sitkImage0, sitkImage1, origin = None, title=None):
    """
    Overlay the orthogonal views of a two 3D volume from the middle of the volume.
    The two volumes must have the same shape. The first volume is displayed in red,
    the second in green.

    Parameters
    ----------
    sitkImage0 : SimpleITK image
        Image to display in red.
    sitkImage1 : SimpleITK image
        Image to display in green.
    origin : array_like, optional
        Origin of the orthogonal views, represented by a point [x,y,z].
        If None, the middle of the volume is used.
    title : str, optional
        Super title of the figure.

    Note:
    On the axial and coronal views, patient's left is on the right
    On the sagittal view, patient's anterior is on the left
    """
    vol0 = sitk.GetArrayFromImage(sitkImage0)
    vol1 = sitk.GetArrayFromImage(sitkImage1)
    if vol0.shape != vol1.shape:
        raise ValueError('The two volumes must have the same shape.')
    if np.min(vol0) < 0 or np.min(vol1) < 0: # Remove negative values - Relevant for the noisy images
        vol0[vol0 < 0] = 0
        vol1[vol1 < 0] = 0
    if origin is None:
        origin = np.array(vol0.shape) // 2

    sh = vol0.shape
    R = img_as_ubyte(vol0/np.max(vol0))
    G = img_as_ubyte(vol1/np.max(vol1))

    vol_rgb = np.zeros(shape=(sh[0], sh[1], sh[2], 3), dtype=np.uint8)
    vol_rgb[:, :, :, 0] = R
    vol_rgb[:, :, :, 1] = G

    fig, axes = plt.subplots(1, 3, figsize=(12, 4))

    axes[0].imshow(vol_rgb[origin[0], ::-1, ::-1, :])
    axes[0].set_title('Axial')

    axes[1].imshow(vol_rgb[::-1, origin[1], ::-1, :])
    axes[1].set_title('Coronal')

    axes[2].imshow(vol_rgb[::-1, ::-1, origin[2], :])
    axes[2].set_title('Sagittal')

    [ax.set_axis_off() for ax in axes]

    if title is not None:
        fig.suptitle(title, fontsize=16)
```

```
def composite2affine(composite_transform, result_center=None):
    """
    Combine all of the composite transformation's contents to form an equivalent affine transformation.
    Args:
        composite_transform (SimpleITK.CompositeTransform): Input composite transform which contains only
                                                            global transformations, possibly nested.
        result_center (tuple,list): The desired center parameter for the resulting affine transformation.
                                    If None, then set to [0,...]. This can be any arbitrary value, as it is
                                    possible to change the transform center without changing the transformation
                                    effect.
    Returns:
        SimpleITK.AffineTransform: Affine transformation that has the same effect as the input composite_transform.

    Source:
        https://github.com/InsightSoftwareConsortium/SimpleITK-Notebooks/blob/master/Python/22_Transforms.ipynb
    """
    # Flatten the copy of the composite transform, so no nested composites.
    flattened_composite_transform = sitk.CompositeTransform(composite_transform)
    flattened_composite_transform.FlattenTransform()
    tx_dim = flattened_composite_transform.GetDimension()
    A = np.eye(tx_dim)
    c = np.zeros(tx_dim) if result_center is None else result_center
    t = np.zeros(tx_dim)
    for i in range(flattened_composite_transform.GetNumberOfTransforms() - 1, -1, -1):
        curr_tx = flattened_composite_transform.GetNthTransform(i).Downcast()
        # The TranslationTransform interface is different from other
        # global transformations.
        if curr_tx.GetTransformEnum() == sitk.sitkTranslation:
            A_curr = np.eye(tx_dim)
            t_curr = np.asarray(curr_tx.GetOffset())
            c_curr = np.zeros(tx_dim)
        else:
            A_curr = np.asarray(curr_tx.GetMatrix()).reshape(tx_dim, tx_dim)
            c_curr = np.asarray(curr_tx.GetCenter())
            # Some global transformations do not have a translation
            # (e.g. ScaleTransform, VersorTransform)
            get_translation = getattr(curr_tx, "GetTranslation", None)
            if get_translation is not None:
                t_curr = np.asarray(get_translation())
            else:
                t_curr = np.zeros(tx_dim)
        A = np.dot(A_curr, A)
        t = np.dot(A_curr, t + c - c_curr) + t_curr + c_curr - c

    return sitk.AffineTransform(A.flatten(), t, c)
```

```
# Callback invoked when the StartEvent happens, sets up our new data.
def start_plot():
    global metric_values, multires_iterations

    metric_values = []
    multires_iterations = []

# Callback invoked when the EndEvent happens, do cleanup of data and figure.
def end_plot():
    global metric_values, multires_iterations

    del metric_values
    del multires_iterations
    # Close figure, we don't want to get a duplicate of the plot latter on.
    plt.close()

# Callback invoked when the IterationEvent happens, update our data and display new figure.
def plot_values(registration_method):
    global metric_values, multires_iterations

    metric_values.append(registration_method.GetMetricValue())
    # Clear the output area (wait=True, to reduce flickering), and plot current data
    clear_output(wait=True)
    # Plot the similarity metric values
    plt.plot(metric_values, 'r')
    plt.plot(multires_iterations, [metric_values[index] for index in multires_iterations], 'b*')
    plt.xlabel('Iteration Number',fontsize=12)
    plt.ylabel('Metric Value',fontsize=12)
    plt.show()

# Callback invoked when the sitkMultiResolutionIterationEvent happens, update the index into the
# metric_values list.
def update_multires_iterations():
    global metric_values, multires_iterations
    multires_iterations.append(len(metric_values))

def command_iteration(method):
    print(
        f"{method.GetOptimizerIteration():3} "
        + f"= {method.GetMetricValue():10.5f} "
        + f": {method.GetOptimizerPosition()}"
    )
```

## Loading and 3D image and ortho view visualization

**Exercise 1**: Load the ImgT1.nii image and visualize its three ortho-views in one plot being the axial, sagittal, and coronal views

```
dir_in = 'data/'
vol_sitk = sitk.ReadImage(dir_in + 'ImgT1.nii')

# Display the volume
imshow_orthogonal_view(vol_sitk, title='T1.nii')
```

## Apply an affine transformation

**Exercise 2**: Write a function `rotation_matrix(pitch, roll, yaw)` which returns the rotation matrix for a given a roll, pitch, yaw. Make a 4x4 affine matrix with a pitch of 25 degrees.

**Exercise 3**: Apply the rotation to the ImgT1.nii around the central point of the volume and save the rotated images as ImgT1_A.nii. Note that the central point is given in physical units (mm) in the World Coordinate System. You can use the next block of code:

An important consideration it is that ITK transforms store the resampling transform/backward mapping transform (fixed to moving image). And then, internally, it applies the inverse of the transform to the moving image. This means that we have to pass the inverse matrix of the one we have defined. This is because the transformation is applied to the moving image and not to the fixed image. It is important to consider when we want to apply the transformation to the fixed image. Note that the inverse or the rotation matrix is the same as the transpose of the rotation matrix, then, when we set the rotation matrix: ```transform.SetMatrix(rot_matrix.T.flatten())```

For a more general transformation matrix (no only rotations involved), you should compute the inverse matrix:
`transform.SetMatrix(np.linealg.inv(rot_matrix).flatten())`

```
# Define the roll rotation in radians
angle = 25  # degrees
pitch_radians = np.deg2rad(angle)

# Create the Affine transform and set the rotation
transform = sitk.AffineTransform(3)

centre_image = np.array(vol_sitk.GetSize()) / 2 - 0.5 # Image Coordinate System
centre_world = vol_sitk.TransformContinuousIndexToPhysicalPoint(centre_image) # World Coordinate System
rot_matrix = rotation_matrix(pitch_radians, 0, 0)[:3, :3] # SimpleITK inputs the rotation and the translation separately

transform.SetCenter(centre_world) # Set the rotation centre
transform.SetMatrix(rot_matrix.T.flatten())

# Apply the transformation to the image
ImgT1_A = sitk.Resample(vol_sitk, transform)

# Save the rotated image
sitk.WriteImage(ImgT1_A, dir_in + 'ImgT1_A.nii')
```

**Exercise 4**: Visualise ImgT1_A.nii in ortho view and show the rotated image.

```
imshow_orthogonal_view(ImgT1_A, title='T1_A.nii')
overlay_slices(vol_sitk, ImgT1_A, title = 'ImgT1 (red) vs. ImgT1_A (green)')
```

## Registration of a moving image to a fixed image

**Exercise 5**: Find the geometrical transformation of the moving image to the fixed image. The moving image is ImgT1_A.nii and the fixed image is ImgT1.nii. The new rotated image is named ImgT1_B.nii and the optimal affine transformation matrix text file is named A1.txt. You can try to modify the metric and optimizer step length.

**The following code is a template for the registration. You can relate it to the figure 1 in the theory note. You can modify it to your needs.**

*If the computing time is excesive, increase the shrink factor.*

```
# Set the registration - Fig. 1 from the Theory Note
R = sitk.ImageRegistrationMethod()

# Set a one-level the pyramid scheule. [Pyramid step]
R.SetShrinkFactorsPerLevel(shrinkFactors = [2])
R.SetSmoothingSigmasPerLevel(smoothingSigmas=[0])
R.SmoothingSigmasAreSpecifiedInPhysicalUnitsOn()

# Set the interpolator [Interpolation step]
R.SetInterpolator(sitk.sitkLinear)

# Set the similarity metric [Metric step]
R.SetMetricAsMeanSquares()

# Set the sampling strategy [Sampling step]
R.SetMetricSamplingStrategy(R.RANDOM)
R.SetMetricSamplingPercentage(0.50)

# Set the optimizer [Optimization step]
R.SetOptimizerAsPowell(stepLength=0.1, numberOfIterations=25)

# Initialize the transformation type to rigid
initTransform = sitk.Euler3DTransform()
R.SetInitialTransform(initTransform, inPlace=False)

# Some extra functions to keep track to the optimization process
# R.AddCommand(sitk.sitkIterationEvent, lambda: command_iteration(R)) # Print the iteration number and metric value
R.AddCommand(sitk.sitkStartEvent, start_plot) # Plot the similarity metric values across iterations
R.AddCommand(sitk.sitkEndEvent, end_plot)
R.AddCommand(sitk.sitkMultiResolutionIterationEvent, update_multires_iterations)
R.AddCommand(sitk.sitkIterationEvent, lambda: plot_values(R))

# Estimate the registration transformation [metric, optimizer, transform]
tform_reg = R.Execute(fixed_image, moving_image)

# Apply the estimated transformation to the moving image
ImgT1_B = sitk.Resample(moving_image, tform_reg)

# Save
sitk.WriteImage(ImgT1_B, dir_in + 'ImgT1_B.nii')
```

**Exercise 6**: Show the ortho-view of the ImgT1_B.nii. Display the optimal affine matrix found. Does it agree with the expected and what is expected? Why? You can use the following snippets of code:

You can get the estimated transformation using the following code:

```
estimated_tform = tform_reg.GetNthTransform(0).GetMatrix() # Transform matrix
estimated_translation = tform_reg.GetNthTransform(0).GetTranslation() # Translation vector
params = tform_reg.GetParameters() # Parameters (Rx, Ry, Rz, Tx, Ty, Tz)
```

You can also convert the transformation to a homogeneous matrix using the following code:

```
def homogeneous_matrix_from_transform(transform):
    """Convert a SimpleITK transform to a homogeneous matrix."""
    matrix = np.zeros((4, 4))
    matrix[:3, :3] = np.reshape(np.array(transform.GetMatrix()), (3, 3))
    matrix[:3, 3] = transform.GetTranslation()
    matrix[3, 3] = 1
    return matrix

matrix_estimated = homogeneous_matrix_from_transform(tform_reg.GetNthTransform(0))
matrix_applied = homogeneous_matrix_from_transform(transform)
```

And store and load the transformation matrix using the following code:

```
tform_reg.WriteTransform(dir_in + 'A1.tfm')
tform_loaded = sitk.ReadTransform(dir_in + 'A1.tfm')
```

**Exercise 7**: By default, SimpleITK uses the fixed image's origin as the rotation center. Change the rotation center to the center of the fixed image and repeat the registration. Compare the results.

Change the rotation center to the center of the image using the following code and repeating the registration (Exercise 5):

```
initTransform = sitk.CenteredTransformInitializer(fixed_image, moving_image, sitk.Euler3DTransform(), sitk.CenteredTransformInitiali
```

## Generate a series of rotated 3D images

**Exercise 8**: Make four rotation matrices that rotate the ImgT1nii in steps of 60 degrees starting from 60 degrees. Apply the rotation to ImgT1.nii, reslice and store the resulting images as ImgT1_60.nii, ImgT1_120.nii etc. Show in ortho-view that the rotations are applied as expected for each new image.

**Exercise 9**: Use ImgT1_120.nii as the fixed image, and the other three rotated images from Exercise 8 as the moving images. Run the registration to find the affine matrix and include the reslicing procedure for each of the moving images. Show in ortho-view the resliced images and describe what the rotation angles are. Save the transforms with the name "Ex9_60.tfm, Ex10_180.tfm, Ex10_240.tfm" Do the rotations agree with those in Exercise 8?

*Note: You will need to change the step length to handle the larger rotations. A too small step would lead to the convergence in a local minimum. A good value may be 10. You may also benefit from modifying the pyramid schedule.*

## Combining a series of affine matrices

Often, we wish to combine affine matrixes from a series of images with different registration but apply reslicing only once. The reason is that every time applying reslicing it introduces blurring, and if the image is registered and resliced at each step in a series of rotations, the final image will become very affected by blurring. This we can avoid by first finding the transformation matrix per registration step, then combining them into one matrix, and then applying the reslicing as the final step to the combined affine transformation.

**Exercise 10**: Use the ImgT1_240.nii as the fixed image and use the ImgT1.nii as the moving image. Make an affine matrix clockwise by combining the estimated transformation and the affine matrix obtained at each rotation step in exercise 10 and apply reslicing. Show in ortho views that the ImgT1.nii after applying the combined affine matrix is registered as expected. Show the combined affine matrix and explains if it applies the expected rotation angle.

```
# Load the transforms from file
tform_60 = ...
tform_180 = ...
tform_240 = ...
tform_0 = ...

# Option A: Combine the transforms using the sitk.CompositeTransform(3) function
# Concatenate - The last added transform is applied first
tform_composite = sitk.CompositeTransform(3)

tform_composite.AddTransform(tform_240.GetNthTransform(0))
tform_composite.AddTransform(tform_180.GetNthTransform(0))
tform_composite.AddTransform(tform_60.GetNthTransform(0))
tform_composite.AddTransform(tform_0.GetNthTransform(0))
# Transform the composite transform to an affine transform
affine_composite = composite2affine(tform_composite, centre_world)

# Option B: Combine the transforms manually through multiplication of the homogeneous matrices
A = np.eye(4)
for i in range(tform_composite.GetNumberOfTransforms()-1,-1,-1):
    tform = tform_composite.GetNthTransform(i)
    A_curr = homogeneous_matrix_from_transform(tform)
    A = np.dot(A_curr, A)

tform = sitk.Euler3DTransform()
tform.SetMatrix(A[:3,:3].flatten())
tform.SetTranslation(A[:3,3])
tform.SetCenter(centre_world)
```

## Robustness in the registration and number of iterations

If the moving image becomes too noisy the registration becomes unstable due to the appealingly many local minima in the cost function - in other words, there exist many sub-optimal solutions to the “optimal” affine matrix. In this case, the registration will be very sensitive to the selection of the hyperparameters such as the step length and the number of iterations. Moreover, depending on the optimizer the estimated affine matrix may be very unstable and will change significantly if we repeat the registration at the same noise level (This is not a critical issue for the Powell optimizer in this problem).

We can increase the noise level in an image by setting the noise standard deviation say to sigma = 200. We use a normal distributed random generator to generate the noise which is added to the image.

```
moving_image_noisy = sitk.AdditiveGaussianNoise(moving_image, mean=0, standardDeviation=200)
imshow_orthogonal_view(moving_image_noisy, title='Moving image with noise')
```

**Exercise 11**: Use the ImgT1.nii as the fixed image and ImgT1_240.nii as the moving image. Increase the noise level of the moving image and register it to the fixed image and repeat the registration at the same noise-level for different step length. For what standard deviation level and step length does the optimization algorithm cannot find the global minimum? Show the ortho-views of the noisy moving image.

*Note: When the noise is added, the optimizer becomes more sensitive to the step length. We suggest to try, at least, standardDeviation=200, and step lengths = [10, 50, 150, 200].*

By using the pyramidal multi-resolution registration strategy, we can make the registration of the noisy moving image to the fixed image more robust. Here we use the Gaussian pyramid procedure where we keep the image resolution (i.e., the shrink factor) in the different steps of the pyramid but introduce blurring by using a Gaussian filter at different levels. At the highest level most blurring is added, and we only see the coarse details in the image to be registered. Then, we go to finer and finer levels of details by reducing the blurring factor. The pyramidal procedure is implemented in the registration function and typically three levels are used and we just set the sigma = [3.0, 1.0, 0.0].

```
R.SetShrinkFactorsPerLevel(shrinkFactors = [2,2,2])
R.SetSmoothingSigmasPerLevel(smoothingSigmas=[3,1,0])
R.SmoothingSigmasAreSpecifiedInPhysicalUnitsOn()
```

**Exercise 12:** Register the noisy moving image using the pyramidal procedure. Try three levels of setting sigma=[3.0, 1.0, 0.0]. Repeat the registration procedure with different step lengths. Does the image registration become more insensitive to the step length? If not try increasing sigma=[5.0, 1.0, 0.0]. Can one use only 2 levels of the pyramid? What do you suggest of sigma values? **Show the optimal affine matrices for each of the repeats to check robustness.**