Notes for Large Scale Drone Perception

Henrik Skov Midtiby

2022-04-27 14:49:32

Contents

Introduction

These notes are written for the Large Scale Drone Perception course taught by Elzbieta Pastucha and Henrik Skov Midtiby at the University of Southern Denmark.

The notes provides an overview of the topics discussed in class. To get more detailed knowledge please refer to the references in the text.

Best regards,

Henrik

Additional literature

Web: Computer Vision: Algorithms and Applications by Richard Szeliski

Chapter 1

Color segmentation

The topic of this chapter is color segmentation, i.e. how to segment an image into different components based on color information on the pixel level. A way of representing colors are needed to do color segmentation, this is the color space, which will be discussed in section 1.1. Given a certain color representation, the next step is to make a decision rule for which colors belong to each of the classes of the segmentation. Some often used decision rules are describes in section 1.2. In section 1.3 we will look at how to implement this in python using the opencv and numpy libraries.

1.1 Color spaces

Digital images consist of pixels arranged in a pattern, usually a rectangular grid. Each pixel contain information about the color of that particular part of the image. How the color of a pixel is represented is denoted the *color space*. There exists many different color spaces, in this chapter the following color spaces will be discussed.

- Red, Green and Blue (RGB)
- Hue, Saturation and Value (HSV)
- CieLAB
- OK LAB

Each color space has some associated benefits and disadvantages.

1.1.1 Red, Green and blue (RGB) color space

The inspiration for the RGB color space, is how the human eyes perceive light. To sense color our eyes are equipped with three different types of *cones*, which each are sensitive to a certain range of the electromagnetic spectrum.

By adding different amounts of red, green and blue light respectively, it is possible to generate nearly all the color that the human eye can perceive[1].

In RGB, the amount of red, green and blue light that should be added to form a color is described using three numbers; often integers in the range $[0 - 255]$.

As humans are more sensitive to variations in light in dark colors, a γ (gamma) value is used to transform stored values into amounts of light using the equation

$$V_{\text{out}} = A \cdot V_{\text{in}}^{\gamma} \tag{1.1}$$

Where V_{in} is the encoded value, V_{out} is the amount of emitted light, A is a scaling factor and γ is the gamma value. The gamma value is usually around 2[2]. The nonlinear gamma correction can be problematic when doing calculations with colors[3].

1.1.2 Hue, Saturation and Value / Lightning

On issue with the RGB color space is that it is very different from the way we usually describe colors, e.g.

- a dark green color
- a vibrant red color
- a pale yellow color

In these cases a color (red, green, blue, yellow, ...) is mentioned along with a description of its brightness and saturation. The HSV color space describes colors in a vary similar way. In the HSV color space a color is described in terms of the following three values[4]

- Hue

[1] Web: RGB color model
[2] Web: Gamma correction
[3] Video: Computer color is broken (4 min)
[4] Web: HSL and HSV

- Saturation
- Value

Hue describe the basic color (red, yellow, green, cyan, blue, magenta) as a number between 0 and 360. Hue is cyclic, which means that the hue values 1 and 359 are close to each other. Saturation is a number between 0 an 255 describing how much of the pure color specified by the hue is present in the color to describe, is the saturation is low, the color is a shade of gray and if it is high the color is clear.

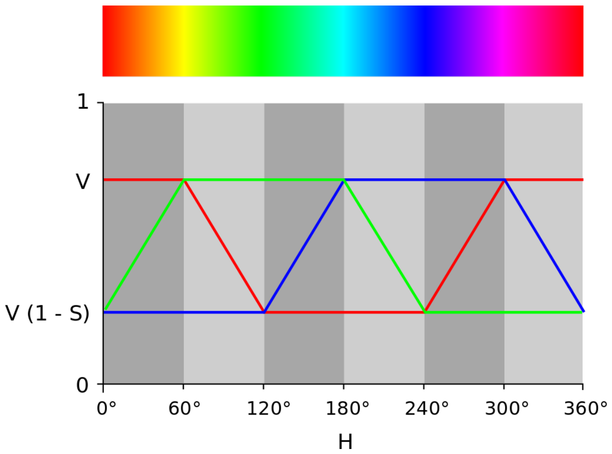

Figure 1.1: *Visual description of how to convert between HSV and RGB color representations. From wikipedia.*

1.1.3 CIE LAB

The CIE LAB color space is an attempt at making a perceptually uniform color space, where distances in the color space equals matches the perceived difference between two colors as a human would interpret it. The three components of the color space are the lightness value L, green/red balance a and the blue/yellow balance b[5].

1.1.4 OK LAB

OK LAB is a brand new color space, that was first described in 2020. The OK LAB color space have better numerical properties and appear more perceptually uniform than CIE LAB[6].
To explore some properties of different color spaces this interactive gradient tool is highly recommended:

- `https://raphlinus.github.io/color/2021/01/18/oklab-critique.html`

[5]Web: CIELAB color space
[6]Web: A perceptual color space for image processing

1.2 Color segmentation

1.2.1 Independent channel thresholds

A basic approach for color based segmentation is to look at each color channel separately. E.g. if we want to see if a color is close to orange . In RGB the orange color is given by the values $(R = 230, G = 179, B = 51)$. A set of requirements for a new color to be classified as orange could be the following

$$200 < R < 255$$
$$149 < G < 209$$
$$21 < B < 81$$

This approach forces the shape of the regions in the color space to accept to have a box shape.

1.2.2 Euclidean distance

A different approach is to look at the the color to classify and the reference color and then calculate a distance between these. Let R_r, G_r and B_r be the reference color value (in RGB color space) and let R_s, G_s and B_s be the color sample that should be classified.
The Euclidean distance can then be calculated using the Pythagorean theorem as follows:

$$\text{distance} = \sqrt{(R_s - R_f)^2 + (G_s - G_r)^2 + (B_s - B_r)^2}$$

If the notation $C_s = [R_s, G_s, B_s]^T$ and $C_r = [R_r, G_r, B_r]^T$, the equation can be written in a more compact form

$$\text{distance} = \sqrt{(C_s - C_r)^T \cdot (C_s - C_r)}$$

The decision rule to accept a color as being close enough to a reference colors can then be written as a requirement on the maximum allowed value of the calculated distance. This decision boundary has the shape of circles in the color space.
To determine the reference color and the threshold, a number of pixels of the object to recognize can be sampled. The reference value can then be set to the mean color value of these pixel and the threshold distance can be set to the maximum distance from the mean color value to the sampled color values.

1.2.3 Mahalanobis distance

To further adapt the decision surface to a set of sampled color values, the Mahalanobis distance can be used[7].

$$\text{distance} = \sqrt{(C_s - C_r)^T \cdot S^{-1} \cdot (C_s - C_r)}$$

[7]Web: Mahalanobis distance

where C_r is the reference color, S is a covariance matrix and C_s is the sample color.
The decision surface generated by the Mahalanobis distance is an ellipsoid in the color space.

1.3 Color segmentation in python

Color segmentation based on independent color channels are implemented in opencv's `inRange` function. The following example demonstrates how to use the `inRange` function

```
filename = "inputfile.jpg"
lower_limits = (130, 130, 100)
upper_limits = (255, 255, 255)
img = cv2.imread(filename)
segmented_image = cv2.inRange(img,
    lower_limits, upper_limits)
```

To extract a sample of pixels from an image, it is effective to annotate a copy of the image with a unique color (e.g. full red) in an image editing program like Gimp. Then the location of the annotated pixels can be determined with `inRange` an image mask can be generated. Given the mask, the function `meanStdDev` can be used to calculate the mean and standard deviation of the color values in the original image.

```
file = "image.jpg"
file_annot = "image_annotated.jpg"
img = cv2.imread(file)
img_annot = cv2.imread(file_annot)
lower_limit = (0, 0, 245)
upper_limit = (10, 10, 256)
mask = cv2.inRange(img_annot,
    lower_limit, upper_limit)
mean, std = cv2.meanStdDev(
    img, mask = mask)
```

As there is no builtin function for calculating the covariance matrix, we need to extract the pixel values into a list (here named pixels) and then weight the observations with the obtained mask. The `reshape` function is used to change the image dimensions to an array with one row per pixel and three columns with the associated color values. Then the average pixel value and the covariance matrix can be found as follows using the `np.cov` and `np.average` functions:

```
pixels = np.reshape(img, (-1, 3))
mask_pixels = np.reshape(mask, (-1))
annot_pix_values = pixels[mask_pixels == 255, ]
avg = np.average(annot_pix_values, axis=0)
cov = np.cov(annot_pix_values.transpose())
```

Often it can be beneficial to perform further analysis of the pixel values (ie. to visualize them). To save the pixel values to a file, this code can be used:

```
np.savetxt("annotated_pixel_values.csv",
        annot_pix_values,
        delimiter=",",
        fmt="%d")
```

To calculate the squared Euclidean distance to a reference color, the following code can be used:

```
shape = pixels.shape
avg_value = np.repeat([avg],
    shape[0], axis=0)
diff = pixels - avg_value
dotproduct = diff * diff
euc_dist = np.sum(dotproduct, axis=1)
euc_dist_image = np.reshape(euc_dist,
        (img.shape[0], img.shape[1]))
```

Similarly the squared Mahalanobis distance can be computed with the code:

```
inv_cov = np.linalg.inv(cov)
moddotproduct = diff * (diff @ inv_cov)
mahalanobis_dist = np.sum(moddotproduct,
    axis=1)
mahalanobis_distance_image = np.reshape(
    mahalanobis_dist,
      (img.shape[0],
       img.shape[1]))
```

1.4 Reference to python and OpenCV

To ensure that you have a proper python environment to work in, the `pipenv` module is recommended

- Web: Pipenv & virtual environments

Some references on how to use OpenCV in python:

- Web: Getting started with images
- Web: Drawing functions in OpenCV
- Web: Basic operations on Images
- Web: Changing Colorspaces
- Web: Image Thresholding

1.5 Getting started exercises

To get access to the support files for these exercises, do the following

- Clone the exercise git repository that you have been given access to on https://gitlab.sdu.dk
- Enter the directory containing the cloned git repository
- Run the command
  `pipenv install`

You should now have all the required dependencies installed. Each group of exercises are placed in a directory. The exercises described in this section are in the `01_getting_started` directory.

Exercise 1.5.1
Put the following content into a file named *draw_on_empty_image.py*

```
import numpy as np
import cv2
img = np.zeros((100, 200, 3), np.uint8)
cv2.line(img, (20, 30), (40, 120),
         (0, 0, 255), 3)
cv2.imwrite("test.png", img)
```

Run the following command on the command line

```
pipenv run python draw_on_empty_image.py
```

If everything worked, there should now be an image named “test.png” in the directory containing a black canvas with a thick red line drawn on top.

Exercise 1.5.2 hint
Load an image into python / opencv, draw something on it and save it again.

Exercise 1.5.3 hint
Load an image and save the three color channels (RGB) in separate files.

Exercise 1.5.4 hint
Load an image and save the upper half of the image in an file.

Exercise 1.5.5 hint
Load an image, convert it to HSV and save the three color channels.

Exercise 1.5.6 hint
Load an image. Locate the brightest pixel in that image and draw a circle around that pixel.

Exercise 1.5.7 hint
Download the image https://commons.wikimedia.org/wiki/File:Daubeny%27s_water_lily_at_BBG_(50824).jpg.
Load the image and generate a pixel mask (bw image) of where the image contains yellow pixels. Save the pixel mask to a file. Experiment with using different colorspaces.

Exercise 1.5.8 hint
Load the image from exercise 1.5.7. Plot the amount of green in each pixel in a single row of the image. Use matplotlib to visualise the data. Repeat this for the two other color channels.

Exercise 1.5.9 hint hint
Load an image. Use matplotlib to visualise a histogram of the pixel intensities in the image.

Exercise 1.5.10
Try to segment the image from exercise 1.5.7 by calculating the euclidean distance to the BGR color $(187, 180, 178)$.

1.6 Counting brightly colored objects

These exercises will be based on some sample images of colored plastic balls on a grass field. You should complete the three exercises with at least one under exposed image, one well exposed and one over exposed image.
To get access to the images, do the following

- Clone the exercise git repository that you have been given access to on https://gitlab.sdu.dk
- Enter the subdirectory
  `02_counting_bright_objects/input`
- Run the command
  `make download_images`

The code snippet shown in figure 1.2 might be handy for debugging the segmentation exercises.

Exercise 1.6.1 hint hint
For each image locate pixels that is a) saturated in all color channels (R = G = B = 255) and b) saturated in at least one color channel (max(R, G, B) = 255) In addition count the number of saturated pixels according to the two measures.

Exercise 1.6.2 hint
We want to count the number of colored balls in the images. As a first step towards that goal, segment the images in the RGB color space. You are only allowed to look at the color channels individually (eg. $R \geq 55$) or look at linear combinations of the color channels ($G - R \geq -10$). Which of the three images are easiest to segment?

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

def compare_original_and_segmented_image(original, segmented, title):
    plt.figure(figsize=(9, 3))
    ax1 = plt.subplot(1, 2, 1)
    plt.title(title)
    ax1.imshow(original)
    ax2 = plt.subplot(1, 2, 2, sharex=ax1, sharey=ax1)
    ax2.imshow(segmented)

img = cv2.imread("input/under_exposed_DJI_0213.JPG")
segmented_image = cv2.inRange(img, (60, 60, 60), (255, 255, 255))
compare_original_and_segmented_image(img, segmented_image, "test")
plt.show()
```

Figure 1.2: *Codesnippet for comparing images next to each other in python, eg. an input image and a segmented image.*

Exercise 1.6.3 hint

Segment the images in the HSV color space. You are only allowed to look at the color channels individually (eg. $H \geq 33$) or look at linear combinations of the color channels ($V - S \geq 0.2$). Which of the three images are easiest to segment?

Exercise 1.6.4 hint

Segment the images in the CieLAB color space. You are only allowed to look at the color channels individually (eg. $H \geq 33$) or look at linear combinations of the color channels ($V - S \geq 0.2$). Which of the three images are easiest to segment?

Exercise 1.6.5 hint

Choose one of the segmented images, filter it with a median filter of an appropriate size and count the number of balls in the image.

Exercise 1.6.6 hint hint

Use the python package `exifread` to extract information about the gimbal pose (yaw, pitch and roll) from one of the images.

Chapter 2

Camera settings

When capturing images there is a set of settings that needs to be chosen. These settings determine can affect the quality of the acquired images and is therefore import to consider.
The image processing pipeline consists of multiple steps, as visualized in figure 2.1. Most courses in machine vision and image analysis will only look at the last step of the pipeline. If you are able to adjust the image acquisition it can make the following image processing much easier.
This chapter will look at the typical camera settings that can be adjusted before or during image acquisition. Some often seen image artefacts will also be discussed.

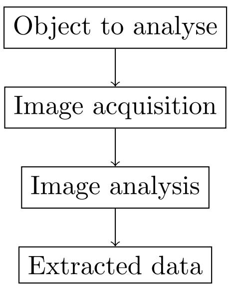

Figure 2.1: *Image processing pipeline*

2.1 A simple image sensor

A digital camera is built around a device that is able to measure properties of the received light. This device is the imaging sensor. Today the Complementary Metal Oxide Sensor (CMOS) is the most commonly used[1]. Earlier the Charge Coupled Device (CCD) was a suitable alternative to CMOS sensors. Before 2020 CCD's provided better signal to noise levels than CMOS sensors, and were preferred in astronomical telescopes.
The imaging sensor is able to count the number of photons that each pixel in the sensor receive. The pixels are arranged in a rectangular pattern. In combination with the lens, this enables the sensor to measure the number of photons coming from different directions[2].
The properties that can be adjusted in a camera system are the following[3] [4]:

- the aperture, which limits the amount of light that reaches the imaging sensor
- the shutter time, in which the sensor counts photons
- the ISO sensor sensitivity, which determines the gain of the sensor, how many photons can be received before the sensor is saturated?

There are also two different ways of reading out data from image sensors, rolling shutter and global shutter. In an imaging sensor with global shutter, the entire image is exposed simultaneously. After exposure all pixel values are read out. This makes it easier to interpret the images and they do not suffer from distortions based on motion of either the camera or the object in the scene. The main disadvantage of global shutter is that the frame rate it reduced compared to a rolling shutter sensor.
In a rolling shutter sensor, one row of pixels is read out at a time. This means that the acquired image contains elements exposed at different times; this causes deformation of objects in motion [5].
Additional resources on rolling and global shutter

- Web: Global & rolling shutter
- Web: Rolling vs Global Shutter

2.1.1 Camera settings when capturing video

Usually try to achieve an open shutter for 50% of the time when capturing video. If much lower, the

[1] Web: Active pixel sensor

[2] Web: What camera measure

[3] Web: A Comprehensive Beginner's Guide to Aperture, Shutter Speed, and ISO

[4] Video: Aperture, Shutter Speed, ISO, & Light Explained ... (14 min)

[5] Video: Why Do Cameras Do This? (Rolling Shutter Explained) (7 min)

motion blur does not match what we will expect and the frames would look choppy[6].

2.1.2 Adjusting camera settings on a DJI drone

Web: DJI Mavic Pro Basic Exposure Settings HELP!!

2.2 Optical filters

Linear polarization can be used to remove reflections from vertical (e.g. a window) or horizontal surfaces (e.g. a sea surface) [7].
A band pass filter can be used to only allow a certain part of the electromagnetic spectrum to reach the sensor. High pass and low pass filters also exist.

2.3 Pinhole camera model

The pinhole camera model is used for mapping 3D world coordinates to 2D image coordinates as follows:

$$s \cdot \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \cdot \big[R \,\big|\, \mathbf{t} \big] \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.1}$$

The pinhole model maps the 3D world coordinate (given in homogeneous coordinates) $(X, Y, Z, 1)^T$ to 2D image coordinates (also in homogeneous coordinates). To compute the mapping, the location and orientation of the camera is used. This is specified as a position vector $\mathbf{t}$ and rotation matrix R of the optical centre of the camera. Some parameters related to the optical system is encoded K matrix. The included parameters cover the focal length in the x and y-directions (f_x and f_y) and the location of the axis of the optical system (c_x and c_y) as well as the parameter describing the skew γ between the x and y axes of the camera sensor. On modern cameras the skew parameter can usually be set to zero.

$$K = \begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Web: Geometry of image formation

2.4 Lens distortion

The pinhole model introduced in 2.3 makes the assumption that all light that reaches the camera sensor passes through a single point – the pin hole. However this is usually replaced with an optical system, consisting of one or more lenses. Such an optical system will introduce distortions to the formed image. These distortions can be divided into two groups

1. radial distortions
2. tangential distortions

The radial distortions are introduced by the optical system, and can be modelled with the equations:

$$x_{\text{distorted}} = x \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.2}$$
$$y_{\text{distorted}} = y \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.3}$$

where r is the distance to the optical axis of the system. The tangential distortions can similarly be modelled with the equations.

$$x_{\text{distorted}} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{2.4}$$
$$y_{\text{distorted}} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{2.5}$$

In most cases the three parameters (k_1, k_2 and k_3) modelling the radial distortions is enough to describe the distortions. If a few cases it could make sense to use additional parameters if the distortion is hard to model.
Web: Camera calibration With OpenCV
Web: Understanding lens distortion
Web: Camera Calibration and 3D Reconstruction – Detailed description

2.5 Camera calibration

Camera calibration is the process of estimating both the parameters in the pinhole camera model and the parameters in the lens distortion model.
Web: Camera calibration using OpenCV

2.6 Projecting observations on to the ground plane

If we look at objects at a known altitude (ie. the sea surface with $Z = 0$) compared to the camera and also know the camera position and orientation, it is possible to calculate the 3D world coordinates of the point we are looking at. So given the image coordinates u and v, the task is to estimate the X and Y parts of the real world coordinates.

$$s \cdot \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \cdot \big[R \,\big|\, \mathbf{t} \big] \cdot \begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} \tag{2.6}$$

[6]Web: Frame rate vs. shutter speed, setting the record straight
[7]Web: Polarizing filter (photography)

2.7 Collection of data

During prototype tests with a B17 bomber in 1935, the pilots failed to configure the plane properly, which caused the plane to crash. To avoid similar problems in the future the company behind the plane, Boing, introduced preflight checklists to ensure that the plane was configured correctly prior to takeoff [8].

When operating UAVs checklists are used to ensure the safety and success of the mission that is to be conducted.

> If you want to read more about checklists and their adoption in aviation this source could be of interest Web: Not flying by the book: Slow adoption of checklists and procedures in WW2 aviation

Exercise 2.7.1
Discuss what actions you would take before taking off with a multirotor UAV. Then make a pre flight checklist for a multirotor UAV.

Exercise 2.7.2
Discuss what actions you would take after landing with a multirotor UAV. Then make a post flight checklist for a multirotor UAV.

Exercise 2.7.3
Collect images with different exposures

- Under exposed images
- Well exposed images
- Over exposed images

Collect images with different orientations of the camera

- Change yaw or pitch

Exercise 2.7.4
Take a look at some of the acquired images (eg. `DJI_0001.JPG` and `DJI_0177.JPG`). Extract information from the image about the position of the UAV during image capture and the orientation of the camera. Try to verify the information by locating known objects in the images.

Exercise 2.7.5 hint
You have acquired a set of images with severe motion blur and have to redo the flight. Which camera settings would you modify for the flight?

Exercise 2.7.6
Take the image DJI_0177.JPG and find information about the camera orientation in the exif data that is embedded in the image. Modify the camera projection script (provided as matlab code) to match the used camera orientation.

Exercise 2.7.7 hint
Project the image from exercise 2.7.6 down on the ground plane.

Exercise 2.7.8
Take multiple images and project them down onto the same ground plane. The images should be taken with the UAV in the same position but with differences in the gimbal orientation.

2.8 Loose ends

- Web: Quick D: Static helicopter blades

[8] Web: Checklist born – B-17 Bomber pre-flight checklist

Chapter 3

Pumpkin counting miniproject

In this mini-project, you will work in groups of up to three students. The project deals with how to estimate the number of pumpkins in a set of UAV images from a single pumpkin field. To estimate this number, you will need to apply several of the methods that we have discussed in the class.
We expect you to hand in a report that describes how your have dealt with the exercises listed below. Please include the following elements in the report:

- a description of the overall goal of the project
- a description of what has been done as part of each exercise
- references to sources of information
- source code for reproducing the results
- example input data
- example output data
- comments on the obtained results

The report should be handed in before Thursday the 17th of March at midnight (23.59) using IT's Learning.
The dataset that should be used for the project can be downloaded from this link: `https://nextcloud.sdu.dk/index.php/s/L8J4iw7FMCTzS2K`
The project is divided into four parts 1) colour based segmentation, 2) counting of objects in a single image, 3) generation of orthomosaics and 4) finally counting of pumpkins in the entire field. The three first parts can be solved more or less independently whereas the fourth part can only be completed when all the other elements are in place.
Ela will be present to help with questions related to the project in the scheduled class session on March 4th.

3.1 Colour based segmentation

This part consists of using colour based information to segment individual images from a pumpkin field. The segmented images should end up having a black background with smaller white objects on top.

Exercise 3.1.1 hint
Annotate some pumpkins in a test image and extract information about the average pumpkin colour in the annotated pixels. Calculate both mean value and standard variation. Use the following two colour spaces: RGB and CieLAB. Finally try to visualise the distribution of colour values.

Exercise 3.1.2
Segment the orange pumpkins from the background using color information. Experiment with the following segmentation methods

1. `inRange` with RGB values
2. `inRange` with CieLAB values
3. Distance in RGB space to a reference colour

Exercise 3.1.3
Choose one segmentation method to use for the rest of the mini-project.

3.2 Counting objects

This part is about counting objects in segmented images and then to generate some visual output that will help you to debug the programs.

Exercise 3.2.1 hint
Count the number of orange blobs in the segmented image.

Exercise 3.2.2 hint
Filter the segmented image to remove noise.

Exercise 3.2.3
Count the number of orange blobs in the filtered image.

Exercise 3.2.4 hint hint
Mark the located pumpkins in the input image. This step is for debugging purposes and to convince others that you have counted the pumpkins accurately.

3.3 Generate an orthomosaic

This part deals with orthorectifying multiple images of the same field into a single carthometric product. Choose proper settings for all below processes, taking into consideration the available computing resources

Exercise 3.3.1
Load data into Metashape.

Exercise 3.3.2
Perform bundle adjustment (align photos) and check results

Exercise 3.3.3
Perform dense reconstruction

Exercise 3.3.4
Create digital elevation model

Exercise 3.3.5
Create orthomosaic

Exercise 3.3.6
Limit orthomosaic to pumpkin field

3.4 Count in orthomosaic

Use the python package `rasterio` to perform operations on the orthomosaic using a tile based approach.

Exercise 3.4.1 hint hint hint
Create code that only loads parts of the orthomosaic.

Exercise 3.4.2 hint
Design tile placement incl. overlaps.

Exercise 3.4.3
Count pumpkins in each tile.

Exercise 3.4.4
Deal with pumpkins in the overlap, so they are only counted once.

Exercise 3.4.5
Determine amount of pumpkins in the entire field.

3.5 Endnotes

Reflect on the conducted work in this miniproject.

Exercise 3.5.1
Determine GSD and size of the image field. What is the average number of pumpkins per area?

Exercise 3.5.2
Reflect on whether the developed system is ready to help a farmer with the task of estimating the number of pumpkins in a field.

Chapter 4

Dealing with videos

A video is a sequence of individual images / frames. The change from one frame to the next is usually quite small. This makes it possible to compress videos effectively. It also makes it possible to track objects from frame to frame. Where a single grey scale image can be represented with an intensity function $I(x, y)$ a greyscale video can be represented with the intensity function $I(x, y, t)$, where x and y are the spatial coordinates of the image and t is the frame number. Colour videos are described using three intensity functions, that each give information about a certain colour channel.
Some of the computer vision tasks involving videos are the following:

object tracking track an object over a number of frames

track camera motion estimate the motion of the camera assuming that the scene is static

background subtraction make a model of how the background typically looks and then use the model to detect deviances from this

shot boundary detection determine when one shot / clip is replaced by another in a video

motion segmentation determine which objects moves together in a video

This chapter will discuss *object tracking* and *background estimation*.

> This video gives an overview of this chapter:
> Video: Analysis of image sequences

4.1 Colour based tracking

The first approach to look at is colour based tracking. The idea is to form a model of the colour of an object, and then use that colour model to locate where in a new frame similar colours appear. The approach that will be examined in section 4.1.2 is histogram backprojection. After locating similar colours, the location of the tracked object should be updated. A simple approach to this is the mean shift algorithm which is described in section 4.1.3.

4.1.1 Histogram

A histogram is a method for describing a distribution of variables. 1D histograms are used in many location, eg. to visualize the exposure of images in a cameras viewfinder. A list of input values should be used to generate a histogram. The first step is to break the range of input values into a number of discrete bins (eg. 10 or 100), each bin covers a range of input values. Next step is then to count the number of input values that end up in each bin. Finally the number of elements in each bin is visualized using a bar plot. Together with the number of input values, the number of discrete bins determine the *roughness* of the generated histogram. The more input values, the more bins can be used. If very few input values (less than 30 or so) end up in each bin, the histogram will be dominated by sampling noise.
The same approach can also be used for describing the relation between two (or more) variables. See an example in figure 4.2. For each of the variables, the input values are put in a number of bins just as for the 1D histogram. If the relation between two values should be visualized and 10 bins are used for each of the variables, it leads to $10 \times 10 = 100$ bins in the 2D histogram. Similar for three values each with 8 discrete levels $8 \times 8 \times 8 = 512$. As the number of values increase, the the total number of bins increase rapidly; this is known as the *curse of dimensionality*. In practice this means that it is difficult to get enough input values to fill the bins with enough values to avoid too much noise on the generated histogram.
Video: Image Histograms - 5 Minutes with Cyrill

4.1.2 Histogram backprojection

The histogram backprojection algorithm takes an image and a histogram as input. For each pixel

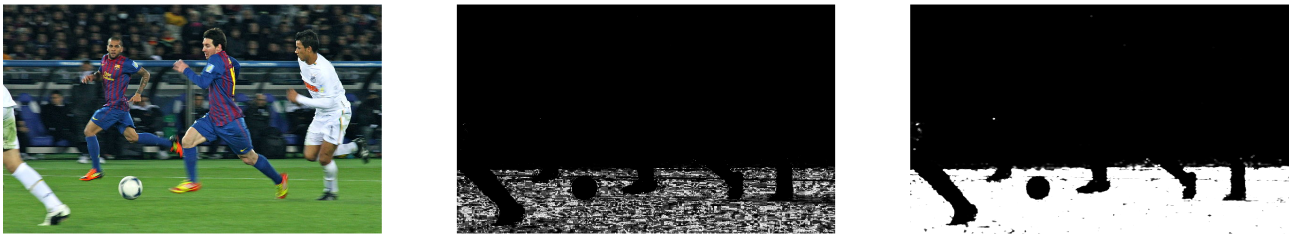

Figure 4.1: *Input image, direct result of histogram backprojection and a dilated version.*

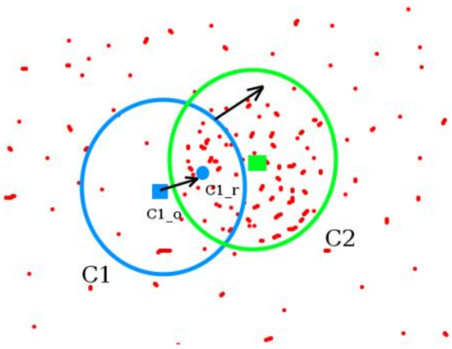

Figure 4.3: *Hill climbing using the mean shift algorithm. From https://docs.opencv.org/master/d7/d00/tutorial_meanshift.html.*

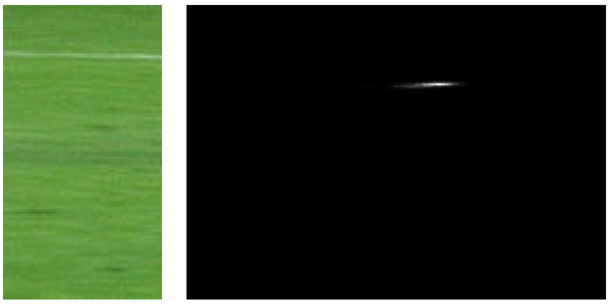

Figure 4.2: *Input colour sample and resulting histogram (with Hue and Saturation as axes). Hue is associated with the y axis and saturation with the x axis.*

in the input image, the corresponding pixel in the generated image is set to the number of values in the corresponding bin in the histogram. The histogram represents the colour distribution that should be matched with the pixels in the input image, regions of the input image that matches the colour distribution will appear brighter in then output image. See an example of histogram backprojection results in figure 4.1.
Web: Histogram – 4: Histogram backprojection

4.1.3 Meanshift and Camshift

After having located pixels with colours similar to a colour model, the next step for colour based tracking is to apply hill climbing on the segmented image. This is implemented in the Meanshift algorithm, which is able to track an object with a known size. To track objects that vary in size the Camshift algorithm should be used instead.
The meanshift algorithm is initiated at the previous location of the tracked object (or if possible at the predicted location of the tracked object). Around the current location, the mean position of pixels (weighted by their intensity) is calculated and the current position is updated to the calculated mean position. This process is repeated until convergence. A single step in this process is illustrated in figure 4.3.
Web: Meanshift and Camshift
Web: Computer Vision Face Tracking For Use in a Perceptual User Interface

4.2 Optical flow

Optical flow is a technique for estimating the motion between two frames in a video. The basic idea is to assume that the objects to track have a constant intensity in the video. Image an object that follows the trajectory specified by the functions $x(t)$ and $y(t)$. The image intensity at the location of the object at t and $t + \Delta t$ should be the same, which is stated in the equation below.

$$I[x(t), y(t), t] = I[x(t + \Delta t), y(t + \Delta t), t + \Delta t]$$

The problem is now to determine where all pixels in the first frame ends up in the second frame given the two images. By taking the limit where $\Delta t \to 0$, the expression can be written as

$$\frac{d}{dt}\left(I[x(t), y(t), t]\right) = 0 \tag{4.1}$$

Applying the chain rule to this expression yields

$$\nabla I \cdot (\dot{x}, \dot{y}) + \frac{\partial}{\partial t} I = 0 \tag{4.2}$$

This requirement must hold for all pixels in the image. The problem is that this approach gives two

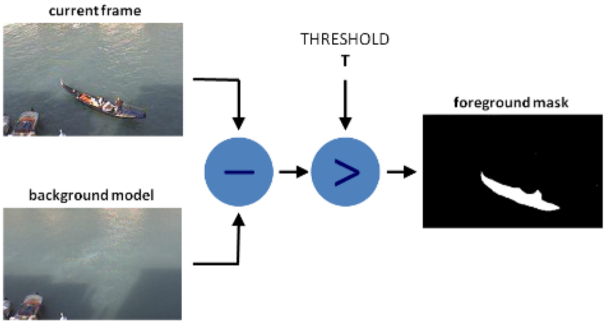

Figure 4.4: From *https://docs.opencv.org/master/d1/dc5/tutorial_background_subtraction.html*

unknowns $\dot{x}$ and $\dot{y}$ and only one equation for each pixel. This is usually dealt with by requiring $\dot{x}$ and $\dot{y}$ to be smooth and only vary slowly [1].
As equation (4.2) only applies in the limit, it works best when the movement of the individual pixels is small. Usually it is first calculated on a scaled down version of the two input images, and then refined on higher resolution images.
OpenCV provides the Lukas Kanade algorithm for tracking the motion of a set of points between two images[2]. For dense optical flow, where the motion of all pixels in an image is tracked, the algorithm by Gunnar Farneback is available in OpenCV[3].
Other approaches to dense optical flow include:
Web: TV-L1 Optical Flow Estimation
Web: A duality based approach for realtime TV–L1 optical flow

> Description of the Lukas Kanade algorithm
> Video: Optical flow

4.3 Background estimation / subtraction

The idea behind background estimation and subtraction is to maintain a model of the background and then compare the current frame with the model of the background. Pixels that differs between the image and the background model is marked as foreground pixels, ie. pixels in motion. This is visualized in figure 4.4.

> This video provides an overview of approaches used for background estimation
> Video: Background estimation

Video: Background Subtraction – Udacity
The simplest approach is to use the previous frame as a model of the background. This is a very rough background model and is only suitable when objects move fast relative to the background. If the objects is moving slowly across the image, only the front and back of the objects will be detected as in motion.
To deal with slowly moving objects, it can be better to look further back in time than the previous frame.
A different approach that is easy to code, is to update the background map with each new frame according to the equation

$$\mathrm{bg}(k+1) = (1-\alpha)\cdot \mathrm{bg}(k) + \alpha \cdot \mathrm{img}(k) \tag{4.3}$$

where α is a weighting factor with a positive value close to zero, eg. 0.05.

[1] Web: Optical flow
[2] Web: Lucas-Kanade Optical Flow in OpenCV
[3] Web: Dense Optical Flow in OpenCV

More advanced schemes can take into account how pixel values vary over time. A Gaussian Mixture Model is implemented in OpenCV. The following background estimators are available in OpenCV

- BackgroundSubtractorMOG
- BackgroundSubtractorMOG2
- BackgroundSubtractorGMG

> For more information, see
>
> - "An Improved Adaptive Background Mixture Model for Real-time Tracking with Shadow Detection", KaewTraKulPong and Bowden 2002

Web: BackgroundSubtractorMOG

4.4 Motion in video exercises

Exercise 4.4.1 hint hint
Watch the video "remember to brake the car.mp4"[4]. Choose two positions in the video (specify image coordinates) which are part of the background during the entire video. Then choose two other positions in the video which changes from background to foreground and back again. Mark all coordinates on the first frame of the video.
Web: Link to video loader example

Exercise 4.4.2
For each of the four locations in the previous exercise, extract the colour value (RGB) for each frame in the video. Visualise / plot how they change over time. The python package matplotlib might be handy for this. What is different between the

[4] Available in the exercise template and here `https://www.youtube.com/watch?v=4i_GFrlaStQ`.

two groups of curves (all background and mixture of background foreground)?

Exercise 4.4.3 hint hint

Make a motion detector by comparing new frames with an average of the earlier seen frames. Show the difference between the current frame and the average. Let the program investigate the entire video.

Exercise 4.4.4

Compare your results with the results from running `BackgroundSubtractorMOG2` on the same video.

Exercise 4.4.5 hint

Try to track the car using the meanshift algorithm from OpenCV.

Exercise 4.4.6

Try to track the car using the camshift algorithm from OpenCV.

Exercise 4.4.7

Track motion in the video by using the Lukas Kanade tracker from OpenCV. The implemented Lukas Kanade algorithm only tracks specified points in the image. Use the `goodFeaturesToTrack` to find points to track.

Exercise 4.4.8

Experiment with the code provided in the file `ex08_dis_dense_optical_flow.py`.

Exercise 4.4.9

Discuss when it is possible to use background estimation methods.

Exercise 4.4.10

Discuss when methods based on dense optical flow are applicable.

Chapter 5

Fiducial markers in images

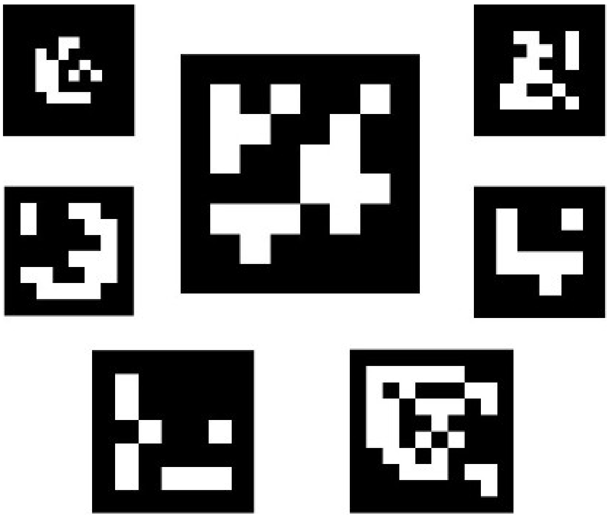

Figure 5.2: *Examples of ArUco markers. From* [*https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html*](https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html)*.*

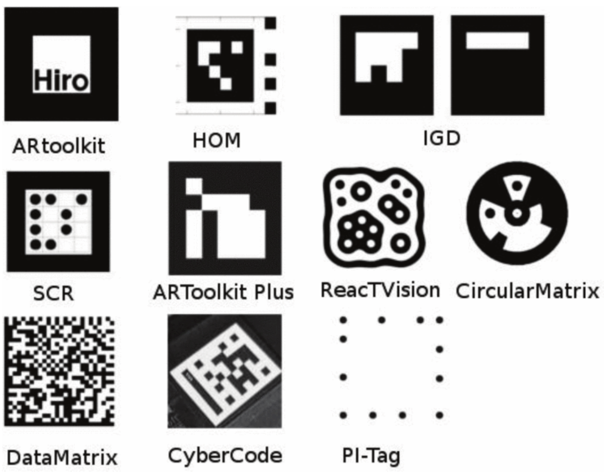

Figure 5.1: *Examples of different fiducial markers. Image from "Comparing fiducial marker systems in the presence of occlusion" by Sagitov et al. 2017.*

Fiducial markers are visual signatures that can be put in a scene, so that they later can be detected and provide information about the location and pose the fiducial marker. Barcodes and QR codes are both examples of fiducial markers, some other examples are shown in figure 5.1. A fiducial marker can help with the following tasks

- Locate a single point
- Locate and identify a single point
- Locate multiple points
- Determine pose of the marker
- Store information

For some markers, it is possible to estimate the pose of a calibrated camera relative to the marker. Fiducial markers have been used for the following tasks:

- camera pose estimation (where is the camera and how is it oriented)
- augmented reality

Fiducial markers can appear in many different forms. They are constructed such that they can be detected and verified automatically. Some fiducial markers indicate a certain location in a scene (which could be augmented with a direction), while other markers contains four or more easily identifiable points which makes it possible to estimate the camera location relative to the marker.
Most fiducial markers contain some amount of information. The amount of information can go from a single integer to a rather long message. An example of a fiducial marker that does not contain any information is the n-fold edge detector, which will be introduced in section 5.2. The well known QR codes can contain up to 4,296 alphanumeric characters.

5.1 The Aruco marker family

The ArUco markers share a black border with a $n \times n$ binary black and white pattern inside the border,

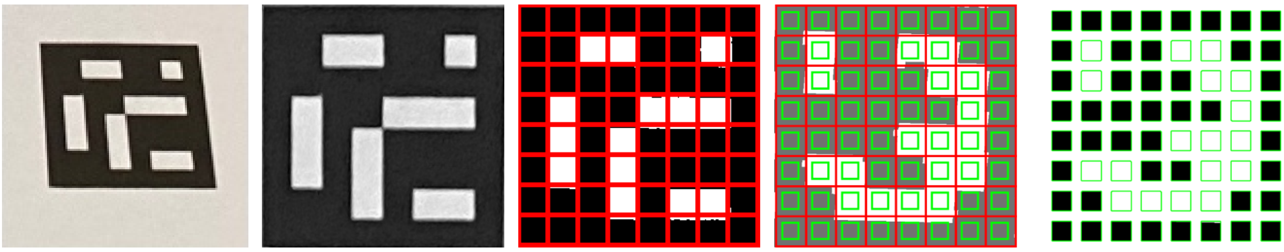

Figure 5.3: *Bit extraction process in ArUco markers. 1) Initial image, 2) perspective corrected image, 3) grid added to binary image and 4) regions that are investigated for pixel values. Images from https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html.*

some examples of ArUco markers are shown in figure 5.2. The binary pattern is used to identify the marker in a given set of possible markers (a dictionary) and also to determine the orientation of the marker[1]. To detect an ArUco Marker, the following process is used

1. perform adaptive thresholding of the image
2. locate contours in the image
3. keep contours that can be approximated well with a polygon with four corners (use the Douglas–Peucker algorithm for this), these are candidate markers
4. locate corners of the candidate markers
5. apply perspective undistortion and then read out the binary pattern
6. look up the binary pattern in the supplied dictionary
7. if a match is found, order the corners according to the orientation of the binary pattern and return the marker
8. otherwise discard the candidate marker

Figure 5.3 illustrates how the stored information is extracted from detected markers.
As all four corners of the ArUco markers are located, it is possible to calculate the camera position relative to the marker. To estimate the camera position, the pinhole camera model is utilized

$$s \cdot \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \cdot \left[R \,\middle|\, \mathbf{t} \right] \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{5.1}$$

where X, Y and Z are the known corner locations of the marker; u and v are the image coordinates of the detected marker; K is the intrinsic camera matrix (which is known from camera calibration). The task is then to determine the rotation matrix R and the location of the camera $\mathbf{t}$, which is done with the Perspective and Point (PnP) algorithm. The OpenCV implementation specifies the rotation matrix in Rodrigues notation, which can be converted to Euler angles.

[1] Xiang Zhang, Fronz, and Navab 2002.

The ArUco markers were introduced in these two papers:

- "Speeded up detection of squared fiducial markers" by Romero-Ramirez, Muñoz-Salinas, and Medina-Carnicer 2018.
- "Automatic generation and detection of highly reliable fiducial markers under occlusion" by Garrido-Jurado et al. 2014.

For more information please consult the opencv tutorial on ArUco makers and the documentation from the researchers behind the system:
Web: Detection of ArUco Markers
Web: ArUco: An efficient library for detection of planar markers and camera pose estimation

Video: Projective 3 Point Algorithm - 5 Minutes with Cyrill
Video: Projective 3-Point Algorithm using Grunert's Method (Cyrill Stachniss)

Inspiration materials:
OpenCV documentation on the Aruco Board class https://docs.opencv.org/3.4/d4/db2/classcv_1_1aruco_1_1Board.html
Peter Klemperer has a brief introduction to using Aruco Boards in 3D http://www.peterklemperer.com/blog/2017/11/30/three-dimensional-aruco-boards/
An opencv tutorial on Aruco Boards https://docs.opencv.org/4.x/db/da9/tutorial_aruco_board_detection.html
Video of how to calibrate camera using opencv and aruco markers https://www.youtube.com/watch?v=SzVutprJ--A

5.2 n-fold edge detector

The n-fold edge detector is able to detect and verify the location of degenerate edge markers. A set of such degenerate edge markers are shown in figure 5.4, the order of a marker is the number of repetitions of black segments when walking around the centre of the marker. The centre of these markers can be detected through convolution with a kernel containing complex values tuned to the number of repetitions of the pattern. What is begin calculated is a specific elements of the Fourier transform applied to the intensity of the image around each point in that image. If a marker with an order that matches the kernel is used for the convolution a strong response is generated. An example is shown in figure 5.5.
Compared to other fiducial markers, the n-fold edge markers contain no information and only mark a single point. However they have one unique property that other fiducial markers lack, they are scale invariant, which means that they can be detected at many different distances. This can be a benefit if you want a UAV to land on a certain location specified by a marker, this enables the UAV to detect the marker at a large distance and the UAV now only have to track that marker during its decent. However it is not possible to estimate the distance to a scale invariant marker, so to estimate the altitude during the descent other sensors must be utilized.

5.3 Exercises about fiducial markers

Please download example images and video by entering the input directory and issuing the command

```
make download_videos
```

Exercise 5.3.1 hint
Clone the git repository `https://github.com/henrikmidtiby/MarkerLocator`. Run the python script `documentation/pythonpic/markertrackerillustrations.py` and investigate what it does.

Exercise 5.3.2 hint hint
Use the `locate_marker` method in `MarkerTracker.py` to locate fourth order markers ($n = 4$) in a fixed image. Visualize the location of the detected marker in the image. As input the image `documentation/pythonpic/input/hubsanwithmarker.jpg` can be used. Examine the data returned by the `locate_marker` method.

Exercise 5.3.3 hint
Download the example video with n-fold-edge markers using the Makefile in the input directory. Use the `locate_marker` method in `MarkerTracker.py` to locate fourth order markers ($n = 4$) in the video file. Visualize the location of the detected marker in the image.

Exercise 5.3.4
Locate a fourth order marker in a video stream from you web cam. Visualize the location of the detected marker in the image.

Exercise 5.3.5 hint
Generate an image with an ARuCo marker by using `cv2.aruco.drawMarker` method. Use a marker from the default `DICT_4x4_250` dictionary.

Exercise 5.3.6
Use `cv2.aruco.detectMarkers` and `cv2.aruco.drawDetectedMarkers` to detect and visualise ARuCo markers in the video `video_with_aruco_markers_dict_4x4_250.mov` which can be found in the input directory.

Exercise 5.3.7
Calibrate your camera and use the calibration parameters to determine the position and orientation of the ARuCo marker. Use the method `cv2.aruco.estimatePoseSingleMarkers`
Calibration targets can be downloaded from this address `https://calib.io/pages/camera-calibration-pattern-generator`.

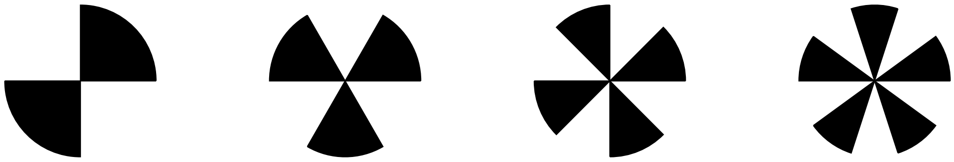

Figure 5.4: *N fold edge markers with different orders ($n = 2 \ldots 5$).*

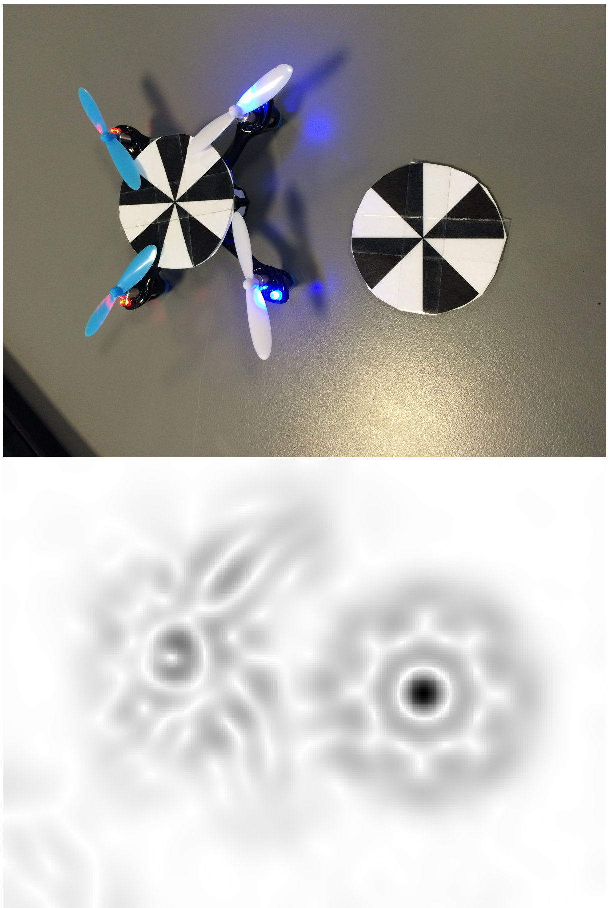

Figure 5.5: *Hubsan with n fold edge marker and the magnitude of the convolution with a fourth order kernel (inverted colours so black denotes a strong response).*

Chapter 6

Feature detectors and video stabilization

Figure 6.1: Image marked with detected SIFT features. Each detected feature is shown with a circle, the radius of the cirle is the size / scale of the detected feature and the feature orientation is marked with a line from the center of the circle.

In the case where you want to locate a known object in a new image or to locate similar objects in two images, image feature detectors and descriptors are the best known solutions. Feature descriptors extracts some of the information present in an image and describes it in a more compact way. The number of pixels in an image is on the order of 10^6, whereas the number of detected features in an image is three orders of magnitude lower at around 10^3.
The purpose of a *feature detector* is to detect points in an image that are easy to locate (they are well defined spatially). Some feature detectors include both a position and a scale for all detected points. Corners and intersection of lines are two examples of things that are easy to locate in an image. Even if a line is well defined and easy to locate, it is not suitable as a feature point as the line does not specify a single point but a set of points that are placed along the line. A simple feature detector is the Harris corner detector[1].
The task of the *feature descriptor* is to describe the area around the detected feature. This description is usually done through a set of numbers. To match two images, features are detected in both images. The detected features are matched based on their descriptors. Such a set of matches can then be used to stitch the two images together.
The following sections will describe some of the often used feature detectors and feature descriptors.

> For more information see chapter 7 in Computer Vision: Algorithms and Applications, 2nd ed.

6.1 Scale Invariant Feature Transform

Given a high resolution image, it is tempting to only look for features at the highest possible level of details with eg. the Harris corner detector. The Harris corner detector only looks inside a small window when detecting features and is therefore not able to detect features that appear on a more coarse scale (eg. if they extent outside of the examined window). One approach to deal with this is to apply the feature detector on downscaled versions of the input image, a so called image pyramid. This is the basic idea behind the Scale Invariant Feature Transform (SIFT) that was introduced in the paper “Object recognition from local scale-invariant features” (D. Lowe 1999). In figure 6.1 the a set of detected features are visualized.
The SIFT feature detector is available in opencv. Given a greyscale image, SIFT features can be located in that image with the following commands:

```
sift = cv2.SIFT_create()
kp = sift.detect(gray)
```

Along with the SIFT feature detector, there is also a SIFT feature descriptor. The feature descriptor looks at the area around the detected feature. The direction of gradients in the image (relative to the

orientation of the detected feature) is calculated and summarized in a histogram. For each keypoint / feature, a 128 element long feature description vector is generated.
Until april 2020 the SIFT feature detector and descriptor was patent protected, which caused them to be absent from the standard opencv installation[2].

- Video: SIFT - 5 Minutes with Cyrill
- Video: SIFT Detector - First Principles of Computer Vision
- Video: Lecture 05 - Scale-invariant Feature Transform (SIFT)
- Web: A short introduction to descriptors
- D.G. Lowe (1999). "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. IEEE, 1150–1157 vol.2. ISBN: 0-7695-0164-8. DOI: `10.1109/ICCV.1999.790410`
- David G. Lowe (Nov. 2004). "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2, pp. 91–110. ISSN: 0920-5691. DOI: `10.1023/B:VISI.0000029664.99615.94`

6.2 Speeded Up Robust Feature transform (SURF)

As the computation of SIFT features are rather time-consuming, work has been done on increasing the speed of feature detection. The Speeded Up Robust Feature (SURF) is such an example. SURF relies on integral images and wavelets for detecting features. Using these tricks, the performance of the feature detector is similar to that of the SIFT feature detector. The feature descriptor is a bit different from the one used in SIFT but is still based on the same principle. Similar to SIFT features, SURF features are rotation invariant. It is however possible to disable the orientation detection and further speed up the feature detection, this variant of the feature detector is named Upright–SURF. This can be beneficial when matching images where the camera orientation is unaltered between the two images.
Due to patent protection, the SURF features are not available in the standard OpenCV distribution. If you choose to compile OpenCV yourself it is possible to enable the SURF features.

- Web: Introduction to SURF (Speeded-Up Robust Features)

6.3 Oriented Briefs

The Oriented BRief (ORB) features were created as a free alternative to SIFT and SURF.
Keypoints are detected using the FAST feature detector. This detector is not scale invariant.

- Web: ORB (Oriented FAST and Rotated BRIEF)
- Web: ORB: An efficient alternative to SIFT or SURF

6.4 Comparison of feature detectors and discriptors

The performance of the discussed feature detectors were investigated by Karami, Prasad, and Shehata 2017. Karami, Prasad, and Shehata finds that matching using ORB features are five to ten times faster than using SIFT features. The cost of the speedup is that match rate is reduced slightly.
What are good and bad properties of the different feature detectors and descriptors

6.5 Image transformations

A 2D image can be transformed to an other 2D image in different ways. This section will look into how the most common used image transformation can be represented by a transformation matrix. The following transformations will be discussed:

- translation
- rotation
- euclidean (translation and rotation)
- affine
- perspective

The five geometric image transformation are show in figure 6.2.

Video: 2d planar image transformations

In the following, the meaning of the variables are as follows

u_1 and v_1: image coordinates in the first image

u_2 and v_2: image coordinates in the first image

x_1, y_1 and z_1: homogeneous image coordinates in the first image

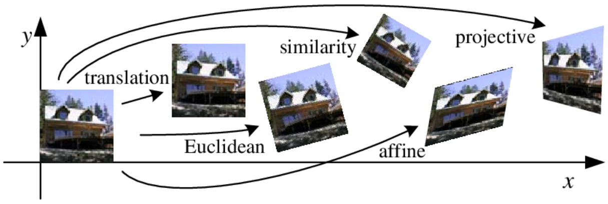

Figure 6.2: *Geometric image transformations. From Computer Vision - Algorithms and Applications by Szeliski.*

x_2, y_2 and z_2: homogeneous image coordinates in the second image

To convert image coordinates from the first image to the associated homogeneous coordinates, a one is added to the coordinate vector

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix}$$

Keep in mind that homogeneous coordinates are considered identical if they only differ by a scale factor s, so the following two homogeneous coordinates refer to the same point

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} s \cdot x_1 \\ s \cdot y_1 \\ s \cdot z_1 \end{pmatrix}$$

In homogeneous coordinates all the listed image transformations can be represented by a transformation matrix with the structure:

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \tag{6.1}$$

The matrix is the projector transformation matrix. To convert from homogenous coordinates to real coordinates, we divide by the z component

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \\ 1 \end{pmatrix} = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix} \tag{6.2}$$

6.5.1 Translation

Translation is represented by the matrix

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a_{13} \\ 0 & 1 & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

The parameters a_{13} and a_{23} represents the shift in x and y coordinates respectively.

6.5.2 Rotation

To rotate an image, the following matrix is used

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

The parameter θ represents the amount of rotation around the origin

6.5.3 Euclidean transformation

The Euclidean transformation consist of both rotation and translation.

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & a_{13} \\ \sin\theta & \cos\theta & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

6.5.4 Similarity

The similarity transform modifies scale, rotation and translation of the image.

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} s \cdot \cos\theta & -s \cdot \sin\theta & a_{13} \\ s \cdot \sin\theta & s \cdot \cos\theta & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

This transform is often used in apps to zoom in and out of an image/map.

6.5.5 Affine

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

6.5.6 Perspective transform

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

The parameters of the perspective transform, can be determined from 4 corresponding points in the image pair (u_1, v_1, u_2 and v_2), by solving a linear system with 8 equations and 8 unknowns. The equations have the form shown below and are converted to linear equations by multiplication with the denominator.

$$u_2 = \frac{a_{11} \cdot u_1 + a_{12} \cdot v_1 + a_{13}}{a_{31} \cdot u_1 + a_{32} \cdot v_1 + 1}$$
$$v_2 = \frac{a_{21} \cdot u_1 + a_{22} \cdot v_1 + a_{23}}{a_{31} \cdot u_1 + a_{32} \cdot v_1 + 1}$$

For one point pair, this gives the two equations.

$$u_2 \cdot (a_{31} \cdot u_1 + a_{32} \cdot v_1 + 1) = a_{11} \cdot u_1 + a_{12} \cdot v_1 + a_{13}$$
$$v_2 \cdot (a_{31} \cdot u_1 + a_{32} \cdot v_1 + 1) = a_{21} \cdot u_1 + a_{22} \cdot v_1 + a_{23}$$

To determine all 8 parameters, 8 or more equations are needed, thus at least four point pairs are needed to determine a perspective transform.

- Video: 2 x 2 image transformations – First Principles Computer Vision
- Video: 2 x 2 image transformations – First Principles Computer Vision
- Web: Homogeneous Coordinates – Cyrill Stachniss

6.6 Filtering matches based on feature descriptors

- Web: How does the Lowe's ratio test work? Stackoverflow answer
- Web: Feature Matching + Homography to find Objects

6.7 Image stabilization

When dealing with video acquired by a drone, the drone tends to move during image acquisition due to e.g. wind gusts. If the field of view is not changing to much, it is possible to stabilize the video using the following approach.

- detect features in a reference image (eg. the first image from the video stream)
- for each new image detect features in that image
- use a geometric image transform to map the new image to the reference image

Notes

[1]Web: Harris Corner Detection – OpenCV
[2]Web: Computer Vision for Busy developers – Describing Features

6.8 Exercises

Exercise 6.8.1 hint hint hint
Load the image `sequence-penguin.jpg` and detect SIFT features in the image. Then draw the detected features on top of the image and save it in the output directory. Please also include the size of the detected marker.

Exercise 6.8.2 hint
Load the image `sequence-cover.jpg` and detect SIFT features in the image. Then draw the detected features on top of the image and save it in the output directory. Please also include the size of the detected marker.

Exercise 6.8.3 hint hint hint
Detect and compute feature descriptors for the two images used in the two previous exercises. Then use the brute force matcher, to locate matches between objects. Finally visualize the matches and comment on what you observe.

Exercise 6.8.4 hint hint
The raw features matches that was visualized in exercise 6.8.3 most likely contains a lot of bad matches. Clean up the matches by using the `findHomography` method and then visualize the matches regarding inliers. In this example the `findHomography` does a really good job of filtering the matches, do you think that it will always perform that well?

Exercise 6.8.5 hint hint
Write the following linear system of equations in matrix form and use `numpy.linalg.solve` to solve the system. Finally verify your solution by inserting it in the original equations.

$$3x + 7y = 3$$
$$2y - 5x = 2$$

Exercise 6.8.6 hint
Apply the similarity transform defined by the matrix

$$\begin{pmatrix} \cos(\pi/6) & \sin(\pi/6) & -50 \\ -\sin(\pi/6) & \cos(\pi/6) & 65 \end{pmatrix}$$

to the point $\begin{pmatrix} 123 & 78 \end{pmatrix}^T$.

Exercise 6.8.7 hint hint
Make a similarity transform, that maps the following point pairs.

Point	x_{img}	y_{img}	x_{obj}	y_{obj}
1	230	1781	100	1600
2	2967	1297	2900	1600

Exercise 6.8.8 hint hint
We often use a similarity transform, when operating tablets. Locate and describe one such usage.

Exercise 6.8.9 hint
Apply the perspective transformation given by the matrix below to the point $\begin{pmatrix} 345 & 234 \end{pmatrix}^T$.

$$\begin{pmatrix} 0.588 & -0.607 & 172 \\ 0.0532 & 0.0772 & 122 \\ 0.000166 & -0.00169 & 1 \end{pmatrix}$$

Exercise 6.8.10 hint

Determine the matrix H describing a perspective transformation from the four matching points given below.

Point	x_{img}	y_{img}	x_{obj}	y_{obj}
1	230	1781	100	1600
2	2967	1297	2900	1600
3	2941	607	2900	100
4	203	59	100	100

A requirement for the H matrix is the following (remember that the positions are in homogeneous coordinates):

$$\begin{pmatrix} 100 \\ 1600 \\ 1 \end{pmatrix} = H \cdot \begin{pmatrix} 230 \\ 1781 \\ 1 \end{pmatrix}$$

You can use the method `findHomography` to verify your result.

Exercise 6.8.11 hint

Experiment with the methods `findHomography` and `warpPerspective`. Try to correct the perspective distortion in an image of a blackboard taken at angle. This image can be used as an example `https://lekmer.dk/images/392530/full.jpg`.

Exercise 6.8.12 hint

Take an image of a piece of A4 paper. Detect the locations of the corners of the paper. Adjust the image for perspective distortions.

Exercise 6.8.13 hint

Try to stabilize the video `../03_image_sequences/input/2016-06-24 Krydset Sdr Bou` by locating features in the first frame and then apply a perspective correction to all following frames that makes the current frame align with the first frame.

Chapter 7

Monocular Visual Odometry

Visual odometry is to estimate the motion of a camera given a sequence of images of the same scene. Usually it is assumed that the camera properties are known in advance (ie. the camera is calibrated). The two main limitations of visual odometry, are that the recovered motion is only given up to an unknown scale and that the orientation of the first frame is also unknown. These issues can be addressed by taking additional sources of information into account (IMU, altimeter and GPS sensors could all help with this). In this chapter, one approach to visual odometry will be described. The approach is inspired by chapter 7 in the SLAM Book[3].

7.1 Algorithm overview

There are different algorithms that implement visual odometry. A simple approach is this suggested by Scaramuzza and Fraundorfer 2011.

1. Capture new frame I_k
2. Extract and match features between I_{k-1} and I_k
3. Compute essential matrix for image pair I_{k-1}, I_k
4. Decompose essential matrix into R_k and t_k, and form T_k
5. Reconstruct 3D points from features matches
6. Compute relative scale and rescale t_k accordingly
7. Concatenate transformation by computing $C_k = C_{k-1}T_k$
8. Repeat from 1.

In the following sections, each step will be investigated closer. To read more about visual odometry, the following tutorial by Davide Scaramuzza is highly recommended:

Davide Scaramuzza and Friedrich Fraundorfer (Dec. 2011). "Visual Odometry: Part I: The First 30 Years and Fundamentals". In: *IEEE Robotics & Automation Magazine* 18.4, pp. 80–92. ISSN: 1070-9932. DOI: 10.1109/MRA.2011.943233

7.2 Location and matching of feature points

Location and matching of features points was discussed in section 6.4.

7.3 Estimation of the essential matrix

The fundamental matrix $\mathbf{F}$ describe the relation between points in two images of the same static scene. If x is a point in the first image and x' is a point in the second image, the following relation holds[4].

$$x^T \; \mathbf{F} \; x' = 0$$

The fundamental matrix has eight degrees of freedom, as it is only defined up to a scale factor. Given eight or more point correspondences, the fundamental matrix can be determined by solving a linear least squares problem.
To estimate the movement of a camera from one image to another of a static scene, the essential matrix $\mathbf{E}$ is easier to work with. The essential matrix related the normalized image coordinates y and y' through the equation:

$$y^T \; \mathbf{E} \; y' = 0 \tag{7.1}$$

The normalized image coordinates are determined from the normal image coordinates and the camera calibration matrix as follows:

$$y = K^{-1}x \tag{7.2}$$

3D punkt: $\vec{Q}$.
Camera 1: $M_1 = K_1 \cdot [I \quad 0]$
Camera 2: $M_2 = K_2 \cdot [R \quad t]$

$$\vec{y}_2^T \cdot E \cdot \vec{y}_1 = \left(R \cdot \vec{Q} - \vec{t}\right)^T \cdot E \cdot \vec{Q}$$

Using $(AB)^T = B^T A^T$

$$= \left(\vec{Q} - \vec{t}\right)^T \cdot R^T \cdot E \cdot \vec{Q}$$

Using $E = R \cdot [t]_x$

$$= \left(\vec{Q} - \vec{t}\right)^T \cdot R^T \cdot R \cdot [t]_x \cdot \vec{Q}$$

Using $R^T \cdot R = I$

$$\begin{aligned} &= \left(\vec{Q} - \vec{t}\right)^T \cdot [t]_x \cdot \vec{Q} \\ &= \vec{Q}^T \cdot [t]_x \cdot \vec{Q} - \vec{t}^T \cdot [t]_x \cdot \vec{Q} \end{aligned}$$

Using properties of the cross product

$$\begin{aligned} &= 0 - 0 \\ &= 0 \end{aligned}$$

The essential matrix carries information about the relative rotation and translation (up to a scale factor) between two camera exposures, this equals five degrees of freedom. There exist algorithms that can determine the essential matrix from as few as five corresponding point pairs and the camera matrix.

7.4 Outlier rejection

When dealing with matched features, it is hard to avoid wrong matches, here denoted outliers. The fraction of outliers can get quite high, and therefore it needs to be dealt with in a smart way. The typical approach is to use the RANdom SAmple Consensus (RANSAC)[5] algorithm, which is used as follows for estimating the essential matrix.

1. five corresponding points are selected among all the matched features
2. the essential matrix is estimated from the selected point pairs
3. all corresponding point pairs are inserted in equation (7.1) and the number of inliers are counted
4. the process is repeated a number of times and the solution with the highest number of inliers is chosen as the best approximation of the essential matrix

7.5 Decomposition of the essential matrix

Given a rotation matrix R and a translation vector t between the first and second camera, the essential matrix can be computed through the equation.

$$\mathbf{E} = \mathbf{R}\,[\mathbf{t}]_\times \tag{7.3}$$

where $[\mathbf{t}]_\times$ is the matrix representation of the cross product with the vector t[6].
This is seen by the following:

$$\begin{aligned} (\tilde{\mathbf{x}}')^T\,\mathbf{E}\,\tilde{\mathbf{x}} &\overset{(1)}{=} (\tilde{\mathbf{x}} - \mathbf{t})^T\,\mathbf{R}^T\,\mathbf{R}\,[\mathbf{t}]_\times\,\tilde{\mathbf{x}} \\ &\overset{(2)}{=} (\tilde{\mathbf{x}} - \mathbf{t})^T\,[\mathbf{t}]_\times\,\tilde{\mathbf{x}} \\ &\overset{(3)}{=} 0 \end{aligned}$$

The essential matrix can be determined from a known rotation and translation of a camera. If the calculation can be reversed, it is possible to estimate the motion of a camera between two images. Given an essential matrix it is in fact possible to estimate the motion, however the motion is not uniquely determined and only up to a scale factor.
Hartley (2004) suggests the following approach to decompose the essential matrix. The first step is to apply singular value decomposition to the essential matrix, this gives a new representation of the essential matrix in the form

$$E = U \cdot S \cdot V^T \tag{7.4}$$

where U and V are orthogonal matrices and S is a diagonal matrix.
The rotation between the two cameras are then given as

$$R_e = U \cdot D \cdot V^T \qquad \text{or} \qquad R_e = U \cdot D^T \cdot V^T$$

where the D matrix is

$$D = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The direction of the translation is given by the expression

$$t = U \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad \text{or} \qquad t = U \cdot \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

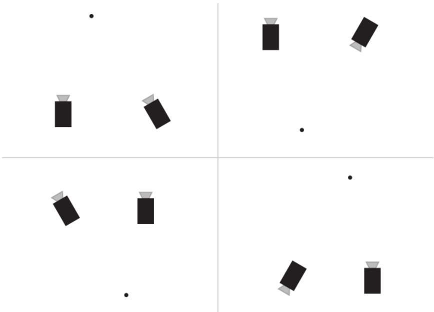

Figure 7.1: Illustration of the four possible camera positions that can be inferred from the essential matrix. Figure from Aanæs 2015

.

With two possibilities for the rotation matrix R and two possibilities for the translation t, there are four combinations in total. The four possible camera arrangements are visualized in figure 7.1[7]. To select the proper camera arrangement, the image correspondences used to estimate essential matrix is used once more, by calculating their associated 3D positions for each of the four camera arrangements. In the proper arrangement the reconstructed image points will lie in front of both cameras.
The method `recoverPose` from OpenCV implements the above described decomposition of the essential matrix and follows with selection of the proper camera arrangement.

7.6 Point triangulation

Given a set of corresponding points between two cameras and the relative motion between the two cameras it is possible to reconstruct the 3D position of the corresponding points. This is called 3D point triangulation[1]. It is implemented in the method `triangulatePoints` in OpenCV.
The `triangulatePoints` method takes the following input values

- a projection matrix from the first camera P_1
- a projection matrix from the second camera P_2
- a list of image coordinates from the first camera x_i^1
- a list of image coordinates from the second camera x_i^2

The structure of the projection matrices are as follows

$$P = K\,[R\ \ t]$$

[1] Henrik Aanæs (2015). *Lecture Notes on Computer Vision*, Section 2.4 Point triangulation

where K is the camera matrix, R is a rotation matrix describing the orientation of the camera and t is a translation vector specifying the location of the camera. The method returns a list of 3D points Q_i, which has the properties

$$x_i^1 = P_1\,Q_i \qquad \text{and} \qquad x_i^2 = P_2\,Q_i$$

7.7 Dealing with small camera motion

Discuss how to handle the case when the camera is not moving (translating) but only rotates. In that case the change between two frames can be explained by a homography.

Make exercise where the reprojection error should be calculated for the essential matrix and for the homography.

7.8 Exercises

Exercise 7.8.1 hint hint
Implement a function that can decompose an essential matrix into the four possible combinations of rotation and translation. As example input this matrix can be used.

$$\begin{pmatrix} -0.00300216 & -0.43213014 & -0.14442965 \\ 0.33859683 & 0.01502989 & -0.60288981 \\ 0.11929845 & 0.54699833 & -0.0246066 \end{pmatrix}$$

One of the decompositions is this

$$R = \begin{pmatrix} 0.98594991 & 0.04674898 & -0.16036614 \\ -0.04766509 & 0.99886163 & -0.00186845 \\ 0.16009624 & 0.00948606 & 0.98705583 \end{pmatrix}$$

$$t = \begin{pmatrix} -0.76471512 \\ 0.20808734 \\ -0.60984461 \end{pmatrix}$$

Exercise 7.8.2 hint hint
Extract and match SIFT features between the images `input/my_photo-1.jpg` and `input/my_photo-2.jpg`. Use the correspondences to estimate the fundamental matrix.

Exercise 7.8.3 hint
Continuation of exercise 7.8.2. Use the output from `cv2.findFundamentalMat` to remove outlier matches that does not fit with the determined fundamental matrix. Visualize the matches before and after the outlier detection.

Exercise 7.8.4 hint

Continuation of exercise 7.8.3. Use the following camera matrix to estimate the essential matrix and remove outliers.

$$\text{cameraMatrix} = \begin{pmatrix} 704 & 0 & 637 \\ 0 & 704 & 376 \\ 0 & 0 & 1 \end{pmatrix}$$

Then compare the number of inliers with the number found in exercise 7.8.3. Which filtering method would you prefer to use? And why?

Exercise 7.8.5 hint

Continuation of exercise 7.8.4 Decompose the determined essential matrix.

Exercise 7.8.6 hint

Continuation of exercise 7.8.5. Determine the 3D location of the matched feature points.

Exercise 7.8.7

Implement the class `ImageAndKeypoints` that when provided with an image, calculates the image features (and descriptors) in the image and stores them for later use. It should be possible to use the class as follows

```
image1 = ImageAndKeypoints(detector)
image1.set_image(self.img1)
image1.detect_keypoints()
```

Exercise 7.8.8

Make a class `ImagePair` that when provided with two instances of `ImageAndKeypoints`, calculates the relative motion of the camera between the two images and estimates the 3D positions of the matched feature points.

Chapter 8

Bundle adjustment with g2o

Bundle adjustment is a complicated optimization process where the spatial location and orientation of cameras and the spatial location of observed points are the optimization parameters. The value that is minimized is usually the reprojection error. Given an observation of the point Q_i by camera j, the pinhole camera model expects to see the corresponding image coordinate of the observation on this coordinate

$$\begin{pmatrix} u_{i,j} \\ v_{i,j} \\ 1 \end{pmatrix} = K\,[R_j|t_j]\;Q_i$$

The reprojection error is the squared difference in the expected image coordinate $(u_{i,j}, v_{i,j})$ and the observed image coordinate $(\hat{u}_{i,j}, \hat{v}_{i,j})$. Given the observed image coordinates, the task is to determine the camera positions t_j, camera orientation R_j and feature point locations Q_i that minimizes the reprojection error.

This is a nonlinear optimization problem, which is usually solved by applying the Levenberg–Marquardt algorithm[1]. Given an initial solution, the Levenberg–Marquardt algorithm is able to improve that solution until the algorithm gets stuck in a local minima. To avoid getting stuck in a bad minima, it is required to start with a decent guess of the set of optimization parameters t_j, R_j and Q_i. Generating such initial guesses was described in section 7.5 and 7.6.

8.1 General Graph Optimization (g2o)

To implement bundle adjustment from scratch is a huge task, which we will not touch upon in this class. Instead the General Graph Optimization (g2o) framework will be used for solving the optimization problem [2].

[1]Web: Levenberg–Marquardt algorithm
[2]Web: g2o - General Graph Optimization

The first step to solve an optimization problem with g2o, is to describe the optimization problem for the g2o framework. The bundle adjustment optimization problem consists of the following elements

- internal camera properties (focal length and optical centre)
- external camera properties (location and orientation)
- interest point locations
- observations that link cameras and interest point location

To simplify matters, the internal camera properties will be fixed. The used camera properties will be taken from an earlier calibration of the used camera. The process of setting up a bundle adjustment problem, will be described in the following.

8.1.1 Data structures for describing the optimization problem

The following three data structures are used to keep track of the elements of the optimization problem:

cameras Store pose and an id of tracked cameras

points Store position and an id of tracked points

observations Store camera id, position id and associated image coordinate

8.1.2 Choosing a proper optimizer

In large bundle adjustment problems, most points will only be seen by a few cameras. This property makes the optimization be a sparse optimization problem and a suitable optimizer should be used as well. The chosen optimizer is able to deal with problems related to motion in a 3D world, this is reflected in the name where the keyword `SE3` (Special Euclidean Group in 3 dimensions) is used.

```
optimizer = g2o.SparseOptimizer()
solver = g2o.BlockSolverSE3(
    g2o.LinearSolverCholmodSE3())
solver = g2o.OptimizationAlgorithmLevenberg(
    solver)
optimizer.set_algorithm(solver)
```

8.1.3 Define camera parameters

```
focal_length = 1000
principal_point = (320, 240)
baseline = 0
cam = g2o.CameraParameters(
    focal_length, principal_point, baseline)
cam.set_id(0)
optimizer.add_parameter(cam)
```

8.1.4 Add cameras to the optimization problem

Each camera will be added as a vertex to the optimization graph. The pose (position and orientation) is specified through the `g2o.SE3Quat` data type, which internally uses quarternions to represent rotations. It is possible to specify the camera as being fixed, if the optimizer is not allowed to change the camera location and pose. A copy of the camera is stored in a dictionary, to make it easier to refer to that specific camera in subsection 8.1.6.

```
camera_vertices = {}
for camera in cameras:
  pose = g2o.SE3Quat(camera.R, camera.t)
  v_se3 = g2o.VertexSE3Expmap()
  v_se3.set_id(camera.camera_id)
  v_se3.set_estimate(pose)
  v_se3.set_fixed(camera.fixed)
  optimizer.add_vertex(v_se3)
  camera_vertices[camera.camera_id] = v_se3
```

8.1.5 Add points to the optimization problem

Each feature point that has been seen by at least two cameras will be added as a vertex to the optimization graph. The point is described by a 3D position. A copy of the point is stored in a dictionary, to make it easier to refer to that specific point in subsection 8.1.6.

```
point_vertices = {}
for point in points:
  vp = g2o.VertexSBAPointXYZ()
  vp.set_id(point.point_id)
  vp.set_marginalized(True)
  point_temp = np.array(
      point.point, dtype=np.float64)
  vp.set_estimate(point_temp)
  optimizer.add_vertex(vp)
  point_vertices[point.point_id]= vp
```

8.1.6 Add observations

The last element that should be added to the optimization problem, is the observations that link the camera poses with the feature point locations. These are specified as edges on the optimization graph.

```
for observation in observations:
  edge = g2o.EdgeProjectXYZ2UV()
  edge.set_vertex(0,
      point_vertices[observation.point_id])
  edge.set_vertex(1,
      camera_vertices[observation.camera_id])
  edge.set_measurement(
      observation.image_coordinates)
  edge.set_information(np.identity(2))
  edge.set_robust_kernel(g2o.RobustKernelHuber())
  edge.set_parameter_id(0, 0)
  optimizer.add_edge(edge)
```

8.1.7 Running the optimizer

When the optimization problem has been defined, the next step is to perform the actual optimization. To run ten iterations of the optimization loop and see some debug information, the following code is used.

```
optimizer.initialize_optimization()
optimizer.set_verbose(True)
optimizer.optimize(10)
```

8.1.8 Extracting problem parameters

After the optimization, the found parameters of the vertices in the optimization problem can be accessed through the `estimate()` method as shown here.

```
for idx, camera in enumerate(self.cameras):
  t = camera_vertices[camera.camera_id].
      estimate().translation()
  self.cameras[idx].t = t
  q = camera_vertices[camera.camera_id].
      estimate().rotation()
  self.cameras[idx].R =
      quarternion_to_rotation_matrix(q)

for idx, point in enumerate(self.points):
  p = point_vertices[point.point_id].estimate()
  self.points[idx].point = np.copy(p)
```

In the last step it is important to make a copy of the estimate of the point vertices using `np.copy`. The vector containing the estimate is allocated by the g2o library, which will free it afterwards. Then the saved value will point towards a freed area of memory.

Web: SLAM Implementation: Bundle Adjustment with g2o

8.2 Useful data structures

When writing python code for visual odometry or similar tasks where non trivial data structures need to be manipulated, I prefer to create my own data types with the `collections.namedtuple`. The idea with a named tuple is that the contents of the tuple can be accessed through a *name* instead of using numerical indexing. Please be aware that tuples and namedTuples are non mutable in python, which means that you cannot change their content. Instead you have to create a new tuple with the modified content.
A small example of how to used named tuples is given here:

```
import collections

# Define the Feature namedtuple
Feature = collections.namedtuple('Feature',
      ['keypoint', 'descriptor', 'feature_id'])

# Create a Feature value
val = Feature([321, 235],
      [1, 2, 3, 4, 5], ‘first feature’)

# Access the values of the feature
print(val.keypoint)
print(val.descriptor)
print(val.feature_id)
```

Below I have shown some example data structures that is relevant for the visual odometry task.

```
import collections

Feature = collections.namedtuple('Feature',
        ['keypoint', 'descriptor', 'feature_id'])

Match = collections.namedtuple('Match',
        ['featureid1', 'featureid2',
            'keypoint1', 'keypoint2',
            'descriptor1', 'descriptor2',
            'distance', 'color'])

Match3D = collections.namedtuple('Match3D',
        ['featureid1', 'featureid2',
            'keypoint1', 'keypoint2',
            'descriptor1', 'descriptor2',
            'distance', 'color',
            'point'])

MatchWithMap = collections.namedtuple(
        'MatchWithMap',
        ['featureid1', 'featureid2',
            'imagecoord', 'mapcoord',
            'descriptor1', 'descriptor2',
            'distance'])
```

The purposes of the above data types are as follows

Feature After detecting features in an image, all detected features are saved in a list of `Feature` objects.

Match When features from two images are matched, the matches are saved in a list of `Match` objects

Match3D After estimating the camera motion, each match object is extended with information about the reconstructed 3D point that is associated with the match.

MatchWithMap When features from a new image is compared with the points in a map, the matches are saved lin a list of `MatchWithMap` objects.

8.3 List comprehensions

In python you often want to do something with each items in a list and then gather all the results in a new list. Often this can be achieved using *list comprehensions* in python.
Web: Python – List Comprehension – w3schools.com

8.4 Exercises

Exercise 8.4.1 hint
Installation of g2o bindings

1. Clone the github repository `https://github.com/RainerKuemmerle/g2o.git`
2. Checkout the pymem branch

   ```
   cd g2o
   git checkout pymem
   ```

3. Install the listed dependencies.

```
sudo apt install cmake libeigen3-dev
sudo apt install libsuitesparse-dev
sudo apt install qtdeclarative5-dev
sudo apt install qt5-qmake
sudo apt install libqglviewer-dev-qt5
```

4. Enable the python bindings and build the library

```
mkdir build
cd build
cmake .. -DG2O_BUILD_PYTHON=ON
make -j 4
```

5. Copy the file `g2o.cpython-38-x86_64-linux-gnu.so` (from the `lib` directory) to the directory where your python scripts are located

6. Launch python and try to import the `g2o` module

Exercise 8.4.2
Extend the `ImagePair` class from exercise 7.8.8 so the estimated camera positions and the detected points are optimized though bundle adjustment.

Exercise 8.4.3
Visualize the location of detected points as well as the camera position. The python library `pangolin` is handy for this.

Exercise 8.4.4
Make a program that loads three images and then estimates the motion of the camera between these three images.
Web: Graph Optimization 1 - Modelling Edges and Vertices
Web: Graph Optimization 4 - g2o introduction - GPS odometry

8.5 Visualization of 3D data with Pangolin

Pangolin is an OpenGL framework that can be used to visualize the obtained 3D point cloud and the associated camera positions.

```
git@github.com:uoip/pangolin.git
cd pangolin
mkdir build
cd build
cmake ..
export CPATH=/usr/include/python3.8:$CPATH
cmake --build .
cd ..
cp pangolin.so path/to/python/project
```

If some errors related to ffmpeg shows up during compilation, that part of the pangolin library can be disabled, as

```
cmake -DBUILD_PANGOLIN_FFMPEG=OFF ..
```

8.6 Interpreting calibration data from metashape

One approach to calibrating a camera is to load some images from the camera into Metashape and then let the program estimate the camera parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<calibration>
  <projection>frame</projection>
  <width>1920</width>
  <height>1080</height>
  <f>1811.0348299321429</f>
  <cx>4.2214536485774854</cx>
  <cy>10.828786855203798</cy>
  <k1>0.23286539919493843</k1>
  <k2>-1.1428125704410166</k2>
  <k3>1.9682032917721508</k3>
  <p1>0.000792256883018114</p1>
  <p2>0.012842139497990459</p2>
  <date>2021-04-21T07:50:37Z</date>
</calibration>
```

The similar calibration data from the tool *camera-calibration-with-large-chessboards* is as follows:

```
Calibration matrix:
[1801.91511542    0.           963.1728924 ]
[   0.          1807.10718836  515.73979394]
[0.  0.  1.]

Distortion parameters (k1, k2, p1, p2, k3):
[[ 2.63750022e-01 -1.55938122e+00
   -5.05944081e-05  1.91358221e-03
   2.88346371e+00]]
```

Value	Metashape	OpenCV
fx	1811.03	1801.92
fy	1811.03	1807.11
cx	4.22	963.17
cy	10.83	515.74
k1	0.232865	0.26375
k2	-1.14	-1.55
k3	1.9682	2.8834
p1	0.000792256	-0.00005059
p2	0.01284	0.001915

Apart from cx and xy the values are reasonably close.

Chapter 9

Visual odometry miniproject

In this mini-project, you will work in groups of up to three students. The project focuses on key elements in a visual odometry / SLAM system.
We expect you to hand in a report that describes how your have dealt with the exercises listed below. Please include the following elements in the report:

- a description of the overall goal of the project
- a description of what has been done
- references to sources of information
- source code for reproducing the results
- example input data
- example output data
- comments on the obtained results

The report should be handed in before Thursday the 12th of May at midnight (23.59) using IT's Learning. The project is divided into four parts 1) examination of dataset, 2) estimation of camera motion between two images, 3) 3d map initialization based on triangulation and 4) tracking of camera motion relative to the generated map.
Henrik will be present to help with questions related to the project in the scheduled class sessions on April 29th and May 6th.
As a base for the project work, you are encouraged to look at the example vision SLAM python project, that has been shared with you on `gitlab.sdu.dk` in your personal repositories for the class.

9.1 Dataset

The dataset that you will work with can be downloaded from nextcloud on this link. The dataset consists of three files, a video file, a logfile from the flight in which the video was recorded and finally a file that contains the parameters of the used camera. The video was recorded with a DJI Phantom 4 Pro. The logfile contains information about recording of two videos, only the first video is of interest in this mini-project.

Exercise 9.1.1 hint
Investigate the contents of the logfile, please pay attention to what values are present in the logfile. Extract the GPS coordinates of the UAV during recording of the first video. Convert the GPS coordinates to UTM and visualize the flight path.

Exercise 9.1.2
Make a python program that goes through all frames in the video one by one and saves every 50th frame to disc. These saved frames will be used in the rest of the project.

Exercise 9.1.3 hint
Discuss the reasons for discarding such a large fraction of the dataset?

9.2 Map initialization

The first step in a visual odometry pipeline is to initialize the map. In this project, the map will be represented as a list of points and their associated feature id's and descriptors. To be able to perform bundle adjustment observations of feature points are also saved as well as a list of the camera poses.
The data structures / classes for representing the 3d points, observations and cameras are given in the provided template. See the definitions of the classes `Map`, `Observation`, `TrackedPoint` and `TrackedCamera`.

Exercise 9.2.1
Choose a feature detector (e.g. SIFT or ORB) and use it to extract features from the first two frames.

Exercise 9.2.2 hint
Match the two sets of features and estimate the essential matrix.

Exercise 9.2.3
For all feature matches, calculate the distance between expected location of a feature point (the epipolar line) and the actual position of the corresponding

feature point in the other image. Provide some summary statistics on the found values (mean, standard deviation, ...).

Exercise 9.2.4
Given the essential matrix and the list of matches, estimate the relative motion of the camera between the two frames.

9.3 3D map initialization

Exercise 9.3.1
Estimate the 3d position of the corresponding image points by triangulation.

Exercise 9.3.2
Add the location of the estimated points to the Map object. Also add the observations that the estimate was based upon. Finally add the cameras to the Map object.

Exercise 9.3.3
Calculate the reprojection error of the current state of the map.

Exercise 9.3.4
Implement bundle adjustment using the g2o python library, to minimize the reprojection error by adjusting the point position estimates and camera poses as given in the `Map` class.

9.4 Estimate new camera positions

The next step in the visual odometry pipeline is to estimate the camera position of new frames. This is achieved by extracting image features from the new image and then search for feature matches with the points in the current map. If a sufficient number of matches is found, the pose of the camera that acquired the frame can be determined.

Exercise 9.4.1
Extract features in a new frame and match the extracted features with the existing map.

Exercise 9.4.2 hint
Use the matches with the map to estimate the camera location and orientation.

Exercise 9.4.3
Add the observation to the map.

Chapter 10

Hints

First hint to 1.5.2 Back to exercise 1.5.2
Use the methods `cv2.imread`, `cv2.imwrite` and `cv2.circle`.

First hint to 1.5.3 Back to exercise 1.5.3
Look at numpy indexing.

First hint to 1.5.4 Back to exercise 1.5.4
Look at numpy indexing.

First hint to 1.5.5 Back to exercise 1.5.5
Look at `cv2.cvtColor`

First hint to 1.5.6 Back to exercise 1.5.6
Look at `cv2.minMaxLoc`

First hint to 1.5.7 Back to exercise 1.5.7
Look at `cv2.inRange`.

First hint to 1.5.8 Back to exercise 1.5.8
An example of how to plot values with matplotlib is given here.

```
import matplotlib.pyplot as plt
plt.plot([8, 3, 6, 2])
filename = "outputfile.png"
plt.savefig(filename)
```

First hint to 1.5.9 Back to exercise 1.5.9
Look at the `plt.hist` from matplotlib.

First hint to 1.6.1 Back to exercise 1.6.1
The `cv2.inRange` function can be used for a).

First hint to 1.6.2 Back to exercise 1.6.2
Open the images in gimp and inspect the color values of the balls with the chosen color.

First hint to 1.6.3 Back to exercise 1.6.3
Open the images in gimp and inspect the color values of the balls with the chosen color.

First hint to 1.6.4 Back to exercise 1.6.4
You will need a tool to convert from RGB to CIELAB values.

First hint to 1.6.5 Back to exercise 1.6.5
199 balls was taken to the field. Do not expect to see them all in an image.

First hint to 1.6.6 Back to exercise 1.6.6
The function `exifread.process_file` is a good place to start.

First hint to 2.7.5 Back to exercise 2.7.5
Exposure time should be decreased while ISO or aperture increased to compensate for the reduction in light on the sensor.

First hint to 2.7.7 Back to exercise 2.7.7
OpenCV hint: Look at the functions: getPerspectiveTransform and warpPerspective.

First hint to 3.1.1 Back to exercise 3.1.1
Information from section 1.3 might be handy.

First hint to 3.2.1 Back to exercise 3.2.1
The `findContours` method can help.

First hint to 3.2.2 Back to exercise 3.2.2
Consider to apply Gaussian blur or a median filter.

First hint to 3.2.4 Back to exercise 3.2.4
I prefer to draw circles on top of each detected pumpkin.

First hint to 3.4.1 Back to exercise 3.4.1

```
# Hint 1: to get image resolution without
# loading it into memory:
import rasterio

filename = "example.tif"
with rasterio.open(filename) as src:
    columns = src.width
    rows = src.height
```

First hint to 3.4.2 Back to exercise 3.4.2
How do you deal with pumpkins on the edge between two tiles?

First hint to 4.4.1 Back to exercise 4.4.1
To download the video from youtube, the python package `youtube-dl` is useful.

First hint to 4.4.3 Back to exercise 4.4.3
To calculate the mean image the following approach can be used. For each frame, add the pixel values

of the frame to an accumulator image. Count the number of frames that have been added to the accumulator. The mean image is now the image in the accumulator divided by the number of frames.

First hint to 4.4.5 Back to exercise 4.4.5
Histogram backprojection can be used to locate the colours that matches the car.

First hint to 5.3.1 Back to exercise 5.3.1
The script will generate several images and save them in the directory `documentation/pythonpic/`

First hint to 5.3.2 Back to exercise 5.3.2
To instantiate the marker tracker, use the following code

```
tracker = MarkerTracker(
            order=4,
            kernel_size=25,
            scale_factor=0.1)
```

First hint to 5.3.3 Back to exercise 5.3.3
To download the data, enter the input subdirectory and issue the command

```
make download_videos
```

First hint to 5.3.5 Back to exercise 5.3.5
Choose the dictionary to use with the command.

```
cv2.aruco.Dictionary_get(cv2.aruco.DICT_4X4_250)
```

First hint to 6.8.1 Back to exercise 6.8.1
To instantiate the SIFT detector use the method `cv2.SIFT_create()`

First hint to 6.8.2 Back to exercise 6.8.2
Look at the hints for exercise 6.8.1.

First hint to 6.8.3 Back to exercise 6.8.3
The method `cv2.SIFT_create().detectAndCompute()` both detects SIFT features and calculates the associated feature descriptors

First hint to 6.8.4 Back to exercise 6.8.4
This tutorial can help with the filtering `https://docs.opencv.org/master/d1/de0/tutorial_py_feature_homography.html`

First hint to 6.8.5 Back to exercise 6.8.5
Rewrite the system of equations in the form $A \cdot \vec{x} = \vec{b}$.

First hint to 6.8.6 Back to exercise 6.8.6
The central calculation is the following

$$\begin{pmatrix} \cos(\pi/6) & \sin(\pi/6) & -50 \\ -\sin(\pi/6) & \cos(\pi/6) & 65 \end{pmatrix} \cdot \begin{pmatrix} 123 \\ 78 \\ 1 \end{pmatrix}$$

First hint to 6.8.7 Back to exercise 6.8.7
The similarity is defined by the following four parameters: s, θ, a_{13} and a_{23}, following the structure used in section 6.5.4.

First hint to 6.8.8 Back to exercise 6.8.8
The similarity transform allows you to move around on a surface.

First hint to 6.8.9 Back to exercise 6.8.9
Remember to use homogeneous coordinates.

First hint to 6.8.10 Back to exercise 6.8.10
Set up eight linear equations. Two equations for each point pair (img to obj).

First hint to 6.8.11 Back to exercise 6.8.11
You can locate the corners of the blackboard manually in Gimp or a similar image editing program.

First hint to 6.8.12 Back to exercise 6.8.12
You can assume that the paper is oriented in portrait mode.

First hint to 6.8.13 Back to exercise 6.8.13
Funktionerne `cv2.findHomography` og `cv2.warpPerspective` er meget nyttige.

First hint to 7.8.1 Back to exercise 7.8.1
To do matrix multiplication in numpy use the `@` operator.

First hint to 7.8.2 Back to exercise 7.8.2
OpenCV provides the method `cv2.findFundamentalMat`.

First hint to 7.8.3 Back to exercise 7.8.3
Investigate the `mask` output from `cv2.findFundamentalMat`.

First hint to 7.8.4 Back to exercise 7.8.4
The more (real) inliers the better.

First hint to 7.8.5 Back to exercise 7.8.5
Use the `recoverPose` method from OpenCV.

First hint to 7.8.6 Back to exercise 7.8.6
Use the method `cv2.triangulatePoints`

First hint to 8.4.1 Back to exercise 8.4.1
There are no direct hints for this exercise, as it is a prerequisite for doing the rest of the exercises.

First hint to 9.1.1 Back to exercise 9.1.1
The `utm` python library can help with the conversion to UTM coordinates: `https://github.com/Turbo87/utm`

First hint to 9.1.3 Back to exercise 9.1.3
At first sight it seems to be a huge waste of information to discard such a large fraction of the frames,

but there is at least one good reason for discarding the frames that is related to algorithm stability.

First hint to 9.2.2 Back to exercise 9.2.2
Consider to filter the key point matches using Lowes rule or through the essential matrix.

First hint to 9.4.2 Back to exercise 9.4.2
The function `solvePnP` is relevant for this exercise. https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html

Second hint to 1.5.9 Back to exercise 1.5.9
The `np.reshape` function can also be handy.

Second hint to 1.6.1 Back to exercise 1.6.1
You can use `cv2.inRange` to find all pixels that are not saturated by using suitable limits. Invert the generated mask to find the partially saturated pixels.

Second hint to 1.6.6 Back to exercise 1.6.6
The `parseString` from `from xml.dom.minidom` is also handy.

Second hint to 3.2.4 Back to exercise 3.2.4
The methods `circle` and `moments` are helpful. See `moments` in action here.

Second hint to 3.4.1 Back to exercise 3.4.1

```
# Hint 2: to load part of an image:
import rasterio
from rasterio.windows import Window

filename = "example.tif"
with rasterio.open(filename) as src:
    # tile ulc - upper left corner,
    # lower left corner... and so on.
    window_location = Window.from_slices(
        (tile.ulc[0], tile.lrc[0]),
        (tile.ulc[1], tile.lrc[1]))
    img = src.read(window=window_location)
```

Second hint to 4.4.1 Back to exercise 4.4.1
I have chosen to show the location of the selected foreground / background pixels in all the frames in the video.

Second hint to 4.4.3 Back to exercise 4.4.3
A different approach is to use an exponential weighted moving average

$$s_t = \alpha x_t + (1 - \alpha)s_{t-1}$$

where x_t is new frame, s_t is the exponentially weighted moving average and α is a learning rate. The learning rate should be in the range $[0.001, 0.1]$.

Second hint to 5.3.2 Back to exercise 5.3.2
Unless the parameter below is set, the quality of the detected marker is probably very low.

```
tracker.track_marker_with_missing_black_leg = False
```

Second hint to 6.8.1 Back to exercise 6.8.1
Use the method `drawKeypoints` to draw the keypoints.

Second hint to 6.8.3 Back to exercise 6.8.3
Instantiate the brute force matcher with the parameters `cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)`.

Second hint to 6.8.4 Back to exercise 6.8.4
The function `findHomography` finds a perspective transformation between two planes.

Second hint to 6.8.5 Back to exercise 6.8.5

$$A = \begin{pmatrix} 3 & 7 \\ -5 & 2 \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

Second hint to 6.8.7 Back to exercise 6.8.7
Set up for linear equations in the four unknowns. Solve these equations.

Second hint to 6.8.8 Back to exercise 6.8.8
The similarity transform also allows you to zoom into objects.

Second hint to 7.8.1 Back to exercise 7.8.1
The method `np.linalg.svd`

Second hint to 7.8.2 Back to exercise 7.8.2
Keep in mind that `cv2.findFundamentalMat` requires that the matched point coordinates are provided as floating point values.

Third hint to 3.4.1 Back to exercise 3.4.1

```
# Hint 3: loaded image has a different shape
# than opencv image, so...

temp = img.transpose(1, 2, 0)
t2 = cv2.split(temp)
img_cv = cv2.merge([t2[2], t2[1], t2[0]]
```

Third hint to 6.8.1 Back to exercise 6.8.1
Try to give the `drawKeypoints` the following flag `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS`

Third hint to 6.8.3 Back to exercise 6.8.3
To draw feature matches, the method `cv2.drawMatches` is helpful.

Bibliography

Aanæs, Henrik (2015). *Lecture Notes on Computer Vision*.

Bishop, Christopher M (2007). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 1st ed. Springer. ISBN: 0387310738. URL: http://www.amazon.com/Pattern-Recognition-Learning-Information-Statistics/dp/0387310738?SubscriptionId=13CT5CVB80YFWJEPWS02&tag=ws&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0387310738.

Csurka, Gabriella et al. (2004). "Visual Categorization with Bags of Keypoints". In: *In Workshop on Statistical Learning in Computer Vision, ECCV*, pp. 1–22. ISBN: 9780335226375. arXiv: arXiv:1210.1833v2. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.604.

Garrido-Jurado, S. et al. (June 2014). "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6, pp. 2280–2292. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2014.01.005.

Hartley, Richard (2004). *Multiple view geometry in computer vision*. Cambridge, UK New York: Cambridge University Press. ISBN: 978-0521540513.

KaewTraKulPong, P. and R. Bowden (2002). "An Improved Adaptive Background Mixture Model for Real-time Tracking with Shadow Detection". In: *Video-Based Surveillance Systems*. Boston, MA: Springer US, pp. 135–144. DOI: 10.1007/978-1-4615-0913-4_11.

Karami, Ebrahim, Siva Prasad, and Mohamed Shehata (Oct. 2017). "Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images". In: *CoRR* abs/1710.0. arXiv: 1710.02726. URL: http://arxiv.org/abs/1710.02726.

Lowe, D.G. (1999). "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. IEEE, 1150–1157 vol.2. ISBN: 0-7695-0164-8. DOI: 10.1109/ICCV.1999.790410.

Lowe, David G. (Nov. 2004). "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2, pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94.

Romero-Ramirez, Francisco J., Rafael Muñoz-Salinas, and Rafael Medina-Carnicer (2018). "Speeded up detection of squared fiducial markers". In: *Image and Vision Computing* 76, pp. 38–47. ISSN: 02628856. DOI: 10.1016/j.imavis.2018.05.004.

Sagitov, Artur et al. (May 2017). "Comparing fiducial marker systems in the presence of occlusion". In: *2017 International Conference on Mechanical, System and Control Engineering (ICMSC)*. IEEE, pp. 377–382. ISBN: 978-1-5090-6530-1. DOI: 10.1109/ICMSC.2017.7959505.

Scaramuzza, Davide and Friedrich Fraundorfer (Dec. 2011). "Visual Odometry: Part I: The First 30 Years and Fundamentals". In: *IEEE Robotics & Automation Magazine* 18.4, pp. 80–92. ISSN: 1070-9932. DOI: 10.1109/MRA.2011.943233.

Sivic and Zisserman (2003). "Video Google: a text retrieval approach to object matching in videos". In: *Proceedings Ninth IEEE International Conference on Computer Vision*. Vol. 2. Iccv. IEEE, 1470–1477 vol.2. ISBN: 0-7695-1950-4. DOI: 10.1109/ICCV.2003.1238663.

Xiang Zhang, S. Fronz, and N. Navab (2002). "Visual marker detection and decoding in AR systems: a comparative study". In: *Proceedings. International Symposium on Mixed and Augmented Reality*. IEEE Comput. Soc, pp. 97–106. ISBN: 0-7695-1781-1. DOI: 10.1109/ISMAR.2002.1115078.