

Notes for Large Scale Drone Perception

Elzbieta Pastucha and Henrik Skov Midtiby

2021-03-15 11:57:49

Contents

1	Color segmentation	4
1.1	Color spaces	4
1.1.1	Red, Green and blue (RGB) color space	4
1.1.2	Hue, Saturation and Value / Lightning	4
1.1.3	CIE LAB	5
1.1.4	OK LAB	5
1.2	Color segmentation	5
1.2.1	Independent channel thresholds	5
1.2.2	Euclidean distance	5
1.2.3	Mahalanobis distance	5
1.3	Color segmentation in python	6
1.4	Reference to python and OpenCV	6
1.5	Getting started exercises	7
1.6	Counting brightly colored objects	7
2	Camera settings	9
2.1	A simple image sensor	9
2.1.1	Camera settings when capturing video	9
2.2	Optical filters	10
2.3	Loose ends	10
3	Pumpkin counting miniproject	11
3.1	Colour based segmentation	11
3.2	Counting objects	11
3.3	Generate an orthomosaic	11
3.4	Count in orthomosaic	12
4	Dealing with videos	13
4.1	Colour based tracking	13
4.1.1	Histogram	13
4.1.2	Histogram backprojection	13
4.1.3	Meanshift and Camshift	14
4.2	Optical flow	14
4.3	Background estimation / subtraction	15
4.4	Motion in video exercises	15

5	Fiducial markers in images	17
5.1	The Aruco marker family	17
5.2	n-fold edge detector	18
5.3	Exercises about fiducial markers	19
6	Hints	20

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,
Ela and Henrik

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 perceive¹.

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

$$V_{\text{out}} = A \cdot V_{\text{in}}^{\gamma} \quad (1.1)$$

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 2². The nonlinear gamma correction can be problematic when doing calculations with colors³.

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 values⁴

- Hue
- Saturation

¹Web: [RGB color model](#)

²Web: [Gamma correction](#)

³Video: [Computer color is broken](#) (4 min)

⁴Web: [HSL and HSV](#)

- 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 and 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.

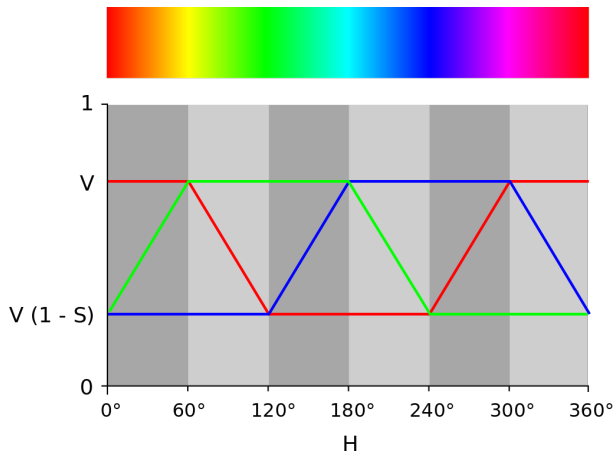


Figure 1.1: Visual description of how to convert between HSV and RGB color representations. *From wikipedia.*

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 LAB⁶.

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>


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

⁵Web: [CIELAB color space](#)

⁶Web: [A perceptual color space for image processing](#)

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:

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

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 used⁷.

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

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.

⁷Web: [Mahalanobis distance](#)

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 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))
cov = np.cov(pixels.transpose(),
    aweights=mask_pixels)
avg = np.average(pixels.transpose(),
    weights=mask_pixels, axis=1)
```

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](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?

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


```

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.

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.

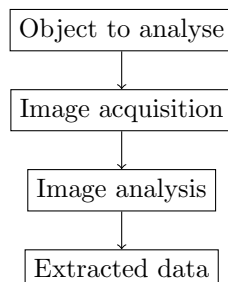


Figure 2.1: Image processing pipeline

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 used¹. 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 directions².

¹Web: [Active pixel sensor](#)

²Web: [What camera measure](#)

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 motion⁵.

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 motion blur does not match what we will expect and the frames would look choppy⁶.

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

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

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

⁶Web: [Frame rate vs. shutter speed, setting the record straight](#)

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 Loose ends

- Web: [Quick D: Static helicopter blades](#)

⁷Web: [Polarizing filter \(photography\)](#)

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 you 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 9th 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.

Both Ela and Henrik will be present to help with questions related to the project in the scheduled class sessions on February 23th and March 2nd.

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 merging multiple images of the same field into a single image (orthomosaic) taken

from a very high altitude.

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.

Exercise 3.4.6

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

Exercise 3.4.7

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 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 his-

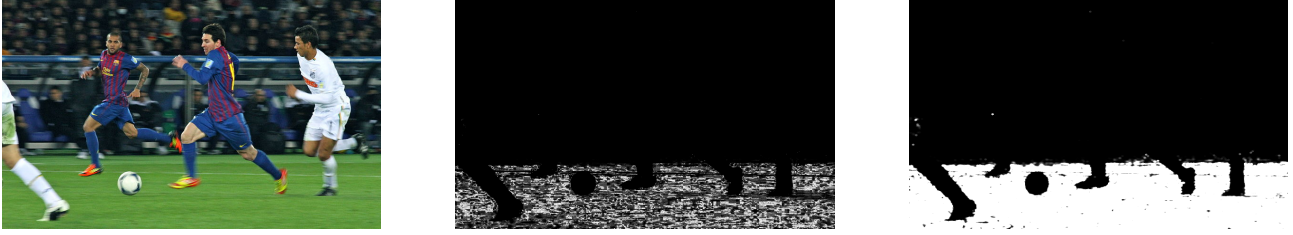


Figure 4.1: Input image, direct result of histogram backprojection and a dilated version.

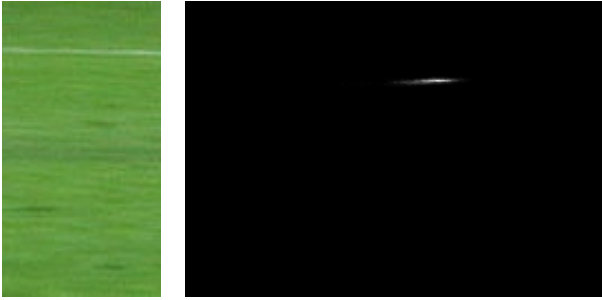


Figure 4.2: Input colour sample and resulting histogram (with Hue and Saturation as axes). Hue is associated with the y axis and saturation with the x axis.

togram 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](#)

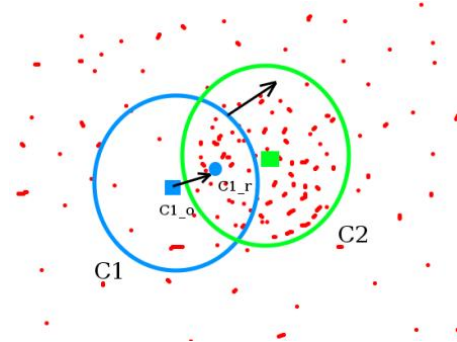


Figure 4.3: Hill climbing using the mean shift algorithm. From https://docs.opencv.org/master/d7/d00/tutorial_meanshift.html.

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 $\Delta t \rightarrow 0$, the expression can be written as

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

Applying the chain rule to this expression yields

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

This requirement must hold for all pixels in the image. The problem is that this approach gives two 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 slowly¹.

As equation (4.2) only applies in the limit, it works best when the movement of the individual pixels is

¹Web: [Optical flow](#)

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 images². For dense optical flow, where the motion of all pixels in an image is tracked, the algorithm by Gunnar Farneback is available in OpenCV³.

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

$$bg(k+1) = (1 - \alpha) \cdot bg(k) + \alpha \cdot img(k) \quad (4.3)$$

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

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

²Web: [Lucas-Kanade Optical Flow in OpenCV](#)

³Web: [Dense Optical Flow in OpenCV](#)

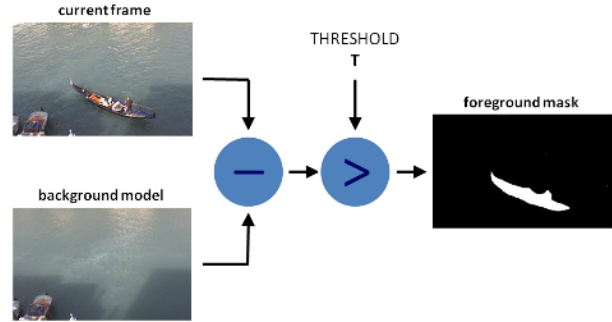


Figure 4.4: From https://docs.opencv.org/master/d1/dc5/tutorial_background_subtraction.html

- BackgroundSubtractorMOG2
- BackgroundSubtractorGMG

For more information, see

- “An Improved Adaptive Background Mixture Model for Real-time Tracking with Shadow Detection”, Kaew-TraKulPong 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 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.

⁴Available in the exercise template and here https://www.youtube.com/watch?v=4i_GFr1aStQ.

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

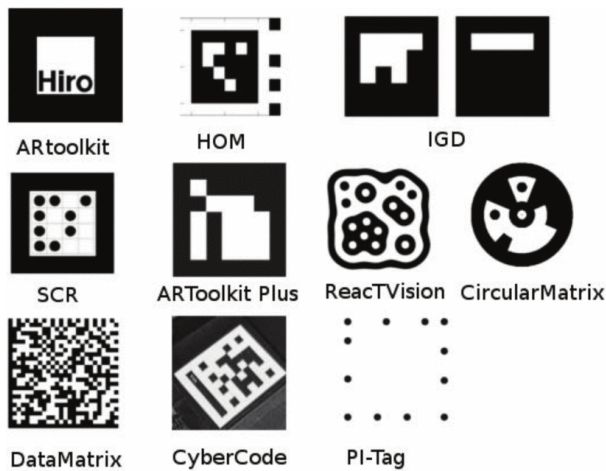


Figure 5.1: Examples of different fiducial markers. Image from “Comparing fiducial marker systems in the presence of occlusion” by Sagitov et al. 2017.

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

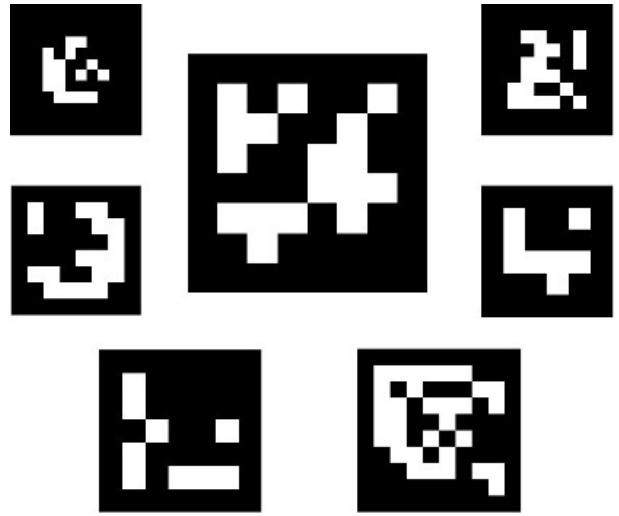


Figure 5.2: Examples of ArUco markers. From https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html.

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, 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

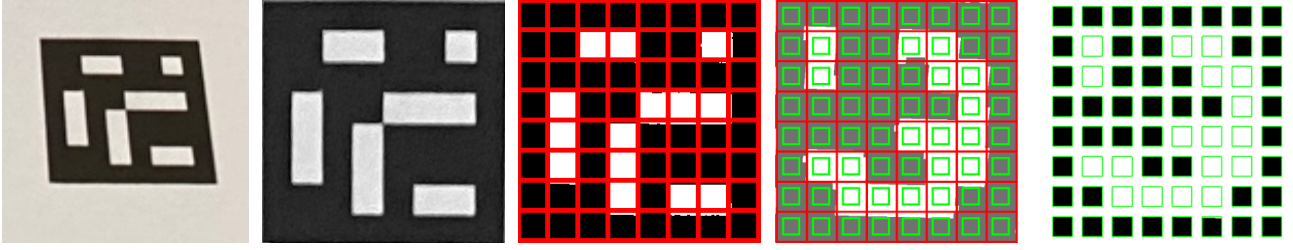


Figure 5.3: Bit extraction process in ArUco markers. 1) Initial image, 2) perspective corrected image, 3) grid added to binary image and 4) regions that are investigated for pixel values. Images from https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html.

marker¹. 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

$$s \cdot \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \cdot [R | \mathbf{t}] \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (5.1)$$

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.

¹Xiang 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](#)

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

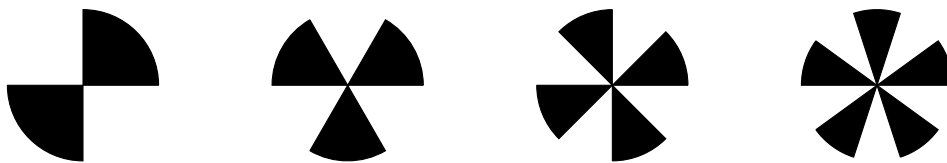


Figure 5.4: N fold edge markers with different orders ($n = 2 \dots 5$).

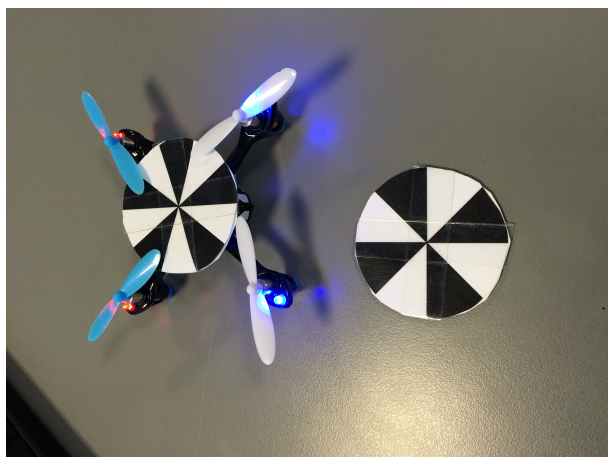


Figure 5.5: Hubsan with n fold edge marker and the magnitude of the convolution with a fourth order kernel (inverted colours so black denotes a strong response).

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>.

Chapter 6

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 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)
```

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 `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 frame 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
```

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]])
```


Bibliography

- 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](https://doi.org/10.1016/j.patcog.2014.01.005).
- 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](https://doi.org/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](https://arxiv.org/abs/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1109/ICMSC.2017.7959505).
- 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](https://doi.org/10.1109/ISMAR.2002.1115078).