# LABexc7-ELE510-2023

October 10, 2023

## 1 ELE510 Image Processing with robot vision: LAB, Exercise 7, Stereo Vision and Camera Calibration.

**Purpose:** *To learn about imaging with two cameras, stereo, and reconstrution by triangulation.*

The theory for this exercise can be found in chapter 13 of the text book [1] and in chapter 4 in the compendium [2]. See also the following documentations for help: - OpenCV - numpy - matplotlib - scipy

**IMPORTANT:** Read the text carefully before starting the work. In many cases it is necessary to do some preparations before you start the work on the computer. Read necessary theory and answer the theoretical part first. The theoretical and experimental part should be solved individually. The notebook must be approved by the lecturer or his assistant.

**Approval:**

The current notebook should be submitted on CANVAS as a single pdf file.

`To export the notebook in a pdf format, goes to File -> Download as -> PDF via LaTeX (.pdf).`

**Note regarding the notebook**: The theoretical questions can be answered directly on the notebook using a *Markdown* cell and LaTex commands (if relevant). In alternative, you can attach a scan (or an image) of the answer directly in the cell.

Possible ways to insert an image in the markdown cell:

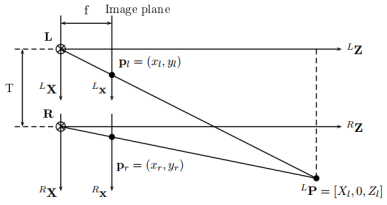
```
![image name]("image_path")
```

```
<img src="image_path" alt="Alt text" title="Title text" />
```

**Under you will find parts of the solution that is already programmed.**

```
<p>You have to fill out code everywhere it is indicated with `...`</p>
<p>The code section under `######## a)` is answering subproblem a) etc.</p>
```

### 1.1 Problem 1 (Correspondence problem)



Assume that we have a simple stereo system as shown in the figure. **L** and **R** denotes the focal point of the Left and Right camera respectively. **P** is a point in the 3D world, and **p** in the 2D image plane. ${}^{L}\mathbf{P}_{w}$ denotes a world point with reference to the focal point of the Left camera. The baseline (line between the two optical centers) is $T = 5\,\text{cm}$ and the focal length $f = 5\,\text{cm}$.

**a)** Consider the scene point ${}^{L}\mathbf{P}_{w} = [0.05\text{m}, 0, 10\text{m}]^{T}$. Suppose that due to various errors, the image coordinate $x_l$ is 2% **bigger** than its true value, while the image coordinate $x_r$ is perfect. What is the error in depth $z_w$, in millimeters (round up to three decimals)?

**b)** An image of resolution $500 \times 500$ pixels is seen by the Left and Right cameras. The image sensor size is 10mm × 10mm. Let the disparity in the image coordinates be up to 25 pixels. Using the same focal point and baseline, what is the depth of the image compare to the cameras?

**c)** Can you explain with your own words the stereo ordering constraint? What is the definition of forbidden zone in this scenario?

**Answers** **a)** First, we would have to calculate the image coordinates of the point ${}^{L}\mathbf{P}_{w}$ in the left and right camera. Since the $y$-value is irrelevant here, it suffices to determine $x_l$ and $x_r$:

$$\frac{x_l}{f} = \frac{{}^{L}\mathbf{P}_{w_x}}{Z_l} \Leftrightarrow x_l = \frac{f \cdot {}^{L}\mathbf{P}_{w_x}}{Z_l} = \frac{0.05\text{m} \cdot 5\text{m}}{10\text{m}} = 0.025\text{m}$$
$$\frac{x_r}{f} = \frac{{}^{L}\mathbf{P}_{w_x} - T}{Z_l} \Leftrightarrow x_r = \frac{f \cdot ({}^{L}\mathbf{P}_{w_x} - T)}{Z_l} = \frac{0.05\text{m} \cdot (5\text{m} - 0.05\text{m})}{10\text{m}} = 0.02475\text{m}$$

*Check: $x_l > x_r$ as expected.*

Since we know $x_l$ is 2 bigger than its true value, we can set $x_l' = 1.02 \cdot x_l = 0.0255\text{m}$. We can then calculate the depth $Z_l'$ as follows:

$$Z_l' = \frac{f \cdot T}{x_l' - x_r'} = \frac{0.05\text{m} \cdot 0.05\text{m}}{0.0255\text{m} - 0.02475\text{m}} = \frac{10}{3}\text{m}$$

So, the error in depth $z_w$ is $|Z_l' - Z_l| = |\frac{10}{3}\text{m} - 10\text{m}| = \frac{20}{3}\text{m} = 6.666\text{m} = 6666.667\text{mm}$.

**b)** Disparity in sensor size:

$$d_{mm} = \frac{10\text{mm}}{500\text{px}} \cdot 25\text{px} = 0.5\text{mm}$$

Depth:

$$Z = \frac{f \cdot T}{d_{mm}} = \frac{0.05\text{m} \cdot 0.05\text{m}}{0.5\text{mm}} = 5\text{m}$$

**c) Stereo Ordering Constraint:** In rectified stereo images, the ordering constraint specifies that the sequence of points along an epipolar line in one image mirrors the sequence of their corresponding points in the other image. This ensures that a point on the left image will find its match to the right on the same epipolar line in the right image, preventing mismatches to the left.

**Forbidden Zone:** The forbidden zone refers to an area in a scene where, if one point is visible to both cameras, another point on the same surface and closer to one camera will be occluded. This region is naturally avoided in stereo matching due to the stereo ordering constraint, which ensures that points from the forbidden zone, lacking correspondence in both images, are not erroneously matched.

### 1.2 Problem 2 (Block Matching)

The simplest algorithm to compute dense correspondence between a pair of stereo images is **block matching**. Block matching is an *area-based* approach that relies upon a statistical correlation between local intensity regions.

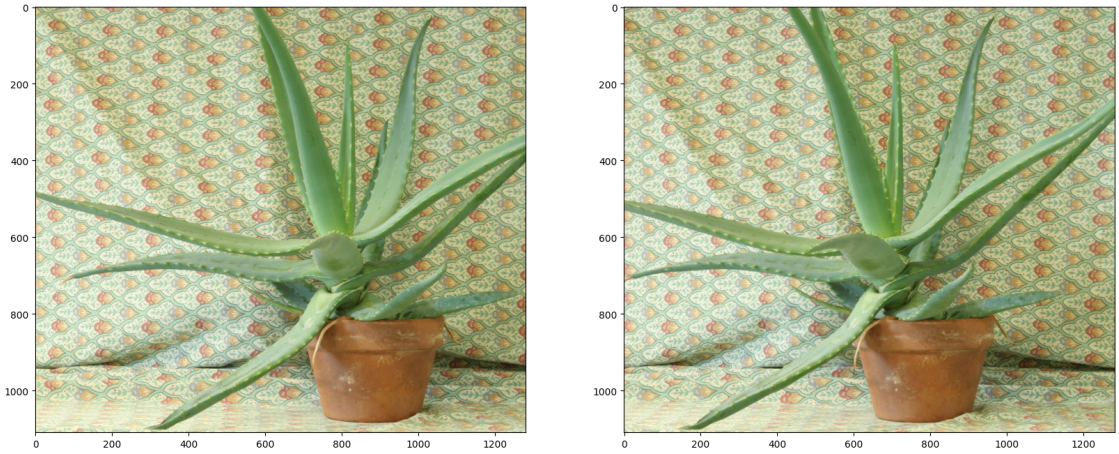
For each pixel (x,y) in the left image, the right image is searched for the best match among all possible disparities $0 \leq d \leq d_{\max}$.

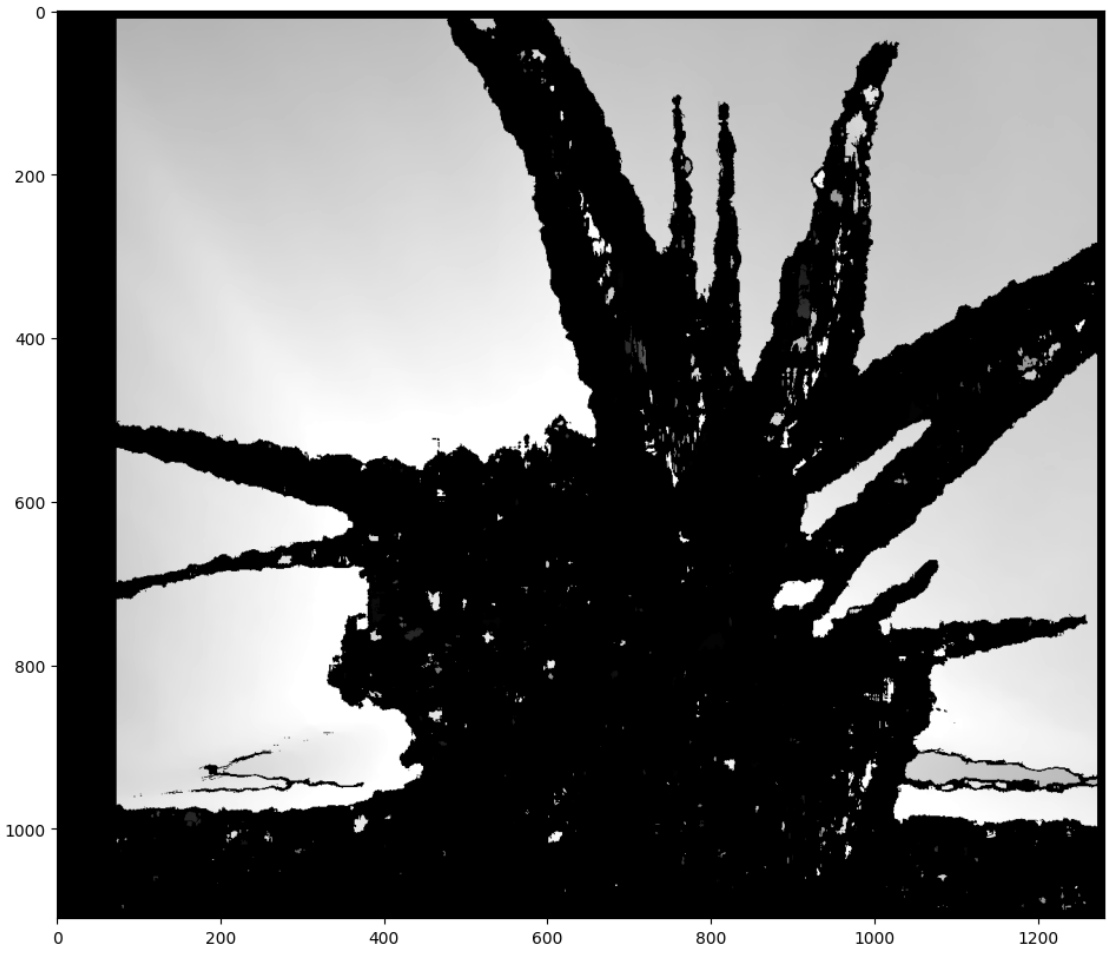
**a)** Use the function `cv2.StereoBM_create(numDisparities=0, blockSize=21)` (Documentation) (Class Documentation) to computing stereo correspondence using the block matching algorithm.

Find the disparity map between the following images: **./images/aloeL.jpg** and **./image/aloeR.jpg**.

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
import cv2

Img_aloeL = cv2.imread('images/aloeL.jpg')
Img_aloeR = cv2.imread('images/aloeR.jpg')

def pipeline(left_image, right_image, *, numDisparities: int = 0, blockSize:␣
 ↪int = 21, draw_original: bool = False):
    # Draw both images side by side
    if draw_original:
        plt.figure(figsize=(20,10))
        plt.subplot(121)
        plt.imshow(cv2.cvtColor(left_image, cv2.COLOR_BGR2RGB))
        plt.subplot(122)
        plt.imshow(cv2.cvtColor(right_image, cv2.COLOR_BGR2RGB))
        plt.show()

    # StearoBM
    stereo_bm = cv2.StereoBM_create(numDisparities=numDisparities,␣
 ↪blockSize=blockSize)
    disparity_bm = stereo_bm.compute(cv2.cvtColor(left_image, cv2.
 ↪COLOR_BGR2GRAY), cv2.cvtColor(right_image, cv2.COLOR_BGR2GRAY))

    # Draw disparity map
    plt.figure(figsize=(20,10))
    plt.imshow(disparity_bm, 'gray')
```

```
    plt.show()

pipeline(Img_aloeL, Img_aloeR, draw_original=True)
```

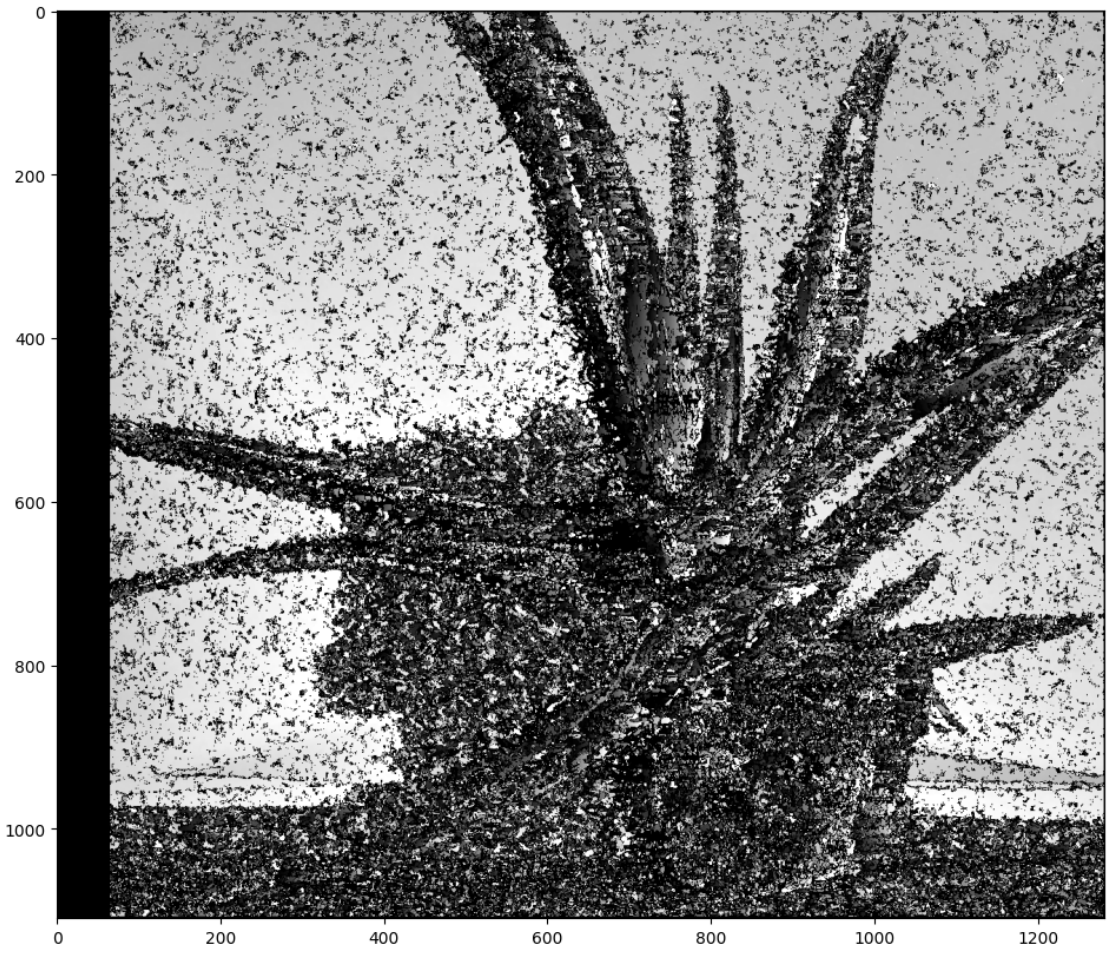


**b)** What happens if you increase the `numDisparities` parameter in the `cv2.StereoBM_create()`? And if you change the `blockSize` parameter?

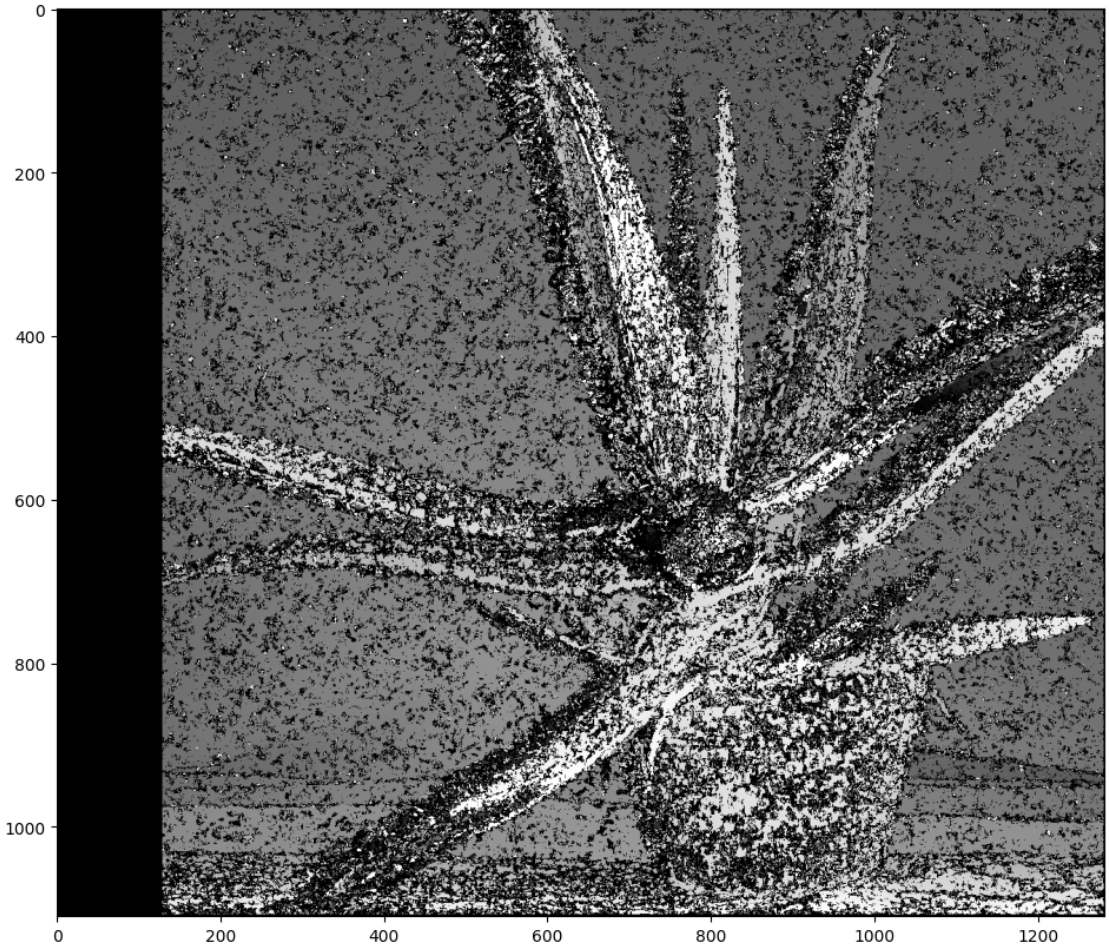
```
[ ]: print("With numDisparities=0 and blockSize=5")
pipeline(Img_aloeL, Img_aloeR, numDisparities=0, blockSize=5)

print("With numDisparities=128 and blockSize=21")
pipeline(Img_aloeL, Img_aloeR, numDisparities=128, blockSize=21)

print("With numDisparities=128 and blockSize=5")
pipeline(Img_aloeL, Img_aloeR, numDisparities=128, blockSize=5)
```

```
With numDisparities=0 and blockSize=5
```




```
With numDisparities=128 and blockSize=21
```


With numDisparities=128 and blockSize=5




When decreasing the block size, we get a lot more artifacts in our disparity map. This intuitively makes sense, since fewer data surrounding a pixel is being considered. Increasing the block size was not possible - cv2 error.

When increasing the number of disparity levels, we can get a lot more detail in our disparity map in the foreground of our image. However, there also seem to be some artifacts here, as edges are being considered a lot closer (darker) than they actually are. This is however inaccurate - the edges of the plant should have roughly the same disparity as the leaves of the plant. It looks more like a cartoonish shader than an actual disparity map.

### 1.3 Problem 3 (Camera calibration)

Calibrate the camera using a set of checkerboard images (you can find them in *./images/left??.jpg*), where `??` indicates the index of the image

Click here for an optional hint

Use the following lines to get the entire list of the images to process:

```
from glob import glob
```

```
img_names = glob('./images/left??.jpg')
```

**a)** Use the checkerboard images to find the feature points using the openCV `cv2.findChessboardCorners()` function (Documentation).

Normally, we have talked about camera calibration as a method to know the intrinsic parameters of the camera, here we want to use the camera matrix and the relative distortion coefficients to undistort the previous images. For a detailed explanation of distortion, read section 13.4.9 of the text book [1].

**b)** Calibrate the camera using the feature points discovered in **a)** and find the relative camera matrix and distortion coefficients using `cv2.calibrateCamera()` function (Documentation).

**P.S.:**
By default, you should find 5 distortion coeffiencients (3 radial distortion coeff. $(k_1, k_2, k_3)$ and 2 tangential coeff. $(p_1, p_2)$); these values are used later to find a new camera matrix and to undistort the images.

**c)** Using the camera matrix and distortion coefficients, transform the images to compensate any kind of distortion using `cv2.getOptimalNewCameraMatrix()` (Documentation) and `cv2.undistort()` (Documentation).

[ ]:
```
# a)
# Function to find the feature points using cv2.findChessboardCorners(...)
# If the function finds the corners, return them, otherwise return None
def findCorners(filename, pattern_size):

    Img = cv2.imread(filename)

    found, corners = cv2.findChessboardCorners(Img, pattern_size)

    if not found: return None
    return corners.reshape(-1, 2)
```

[ ]:
```
# b)
# Function to calibrate the camera.
# Return the camera matrix and the distortion coeffiecients (radial &␣
 ↪tangential)
def calibrateTheCamera(obj_points, img_points, img_shape):

    # The function estimates the intrinsic camera parameters and extrinsic␣
 ↪parameters for each of the views
    ret, camera_matrix, dist_coeffs, _, _ = cv2.calibrateCamera(obj_points,␣
 ↪img_points, img_shape, None, None)

    if not ret:
        print("Calibration failed!")
        return None, None
```

```
    return camera_matrix, dist_coeffs
```

[ ]:
```
# c)
# Function that undistort the images using cv2.getOptimalNewCameraMatrix(...)
 ↪and cv2.undistort(...)
# Plot the new undistorted images.
def undistortImage(filename, camera_matrix, dist_coefs):
    img = cv2.imread(filename,0)
    h, w = img.shape[:2]

    # Returns the new camera intrinsic matrix based on the camera matrix and
 ↪the distortion coefficients
    new_camera_matrix, roi = cv2.getOptimalNewCameraMatrix(camera_matrix,
 ↪dist_coefs, (w, h), 1, (w, h))

    # Transforms an image to compensate for lens distortion using the camera
 ↪matrix,
    # the distortion coefficients and the camera matrix of the distorted image.
    undistorted_img = cv2.undistort(img, camera_matrix, dist_coefs, None,
 ↪new_camera_matrix)

    plt.figure(figsize=(30,30))
    plt.imshow(cv2.cvtColor(undistorted_img, cv2.COLOR_BGR2RGB))
    plt.show()
```

[ ]:
```
from glob import glob

obj_points = []
img_points = []
img_names = glob("images/left??.jpg")


# From the documentation of cv2.findChessboardCorners:
# patternSize - Number of inner corners per a chessboard row and column
#( patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows)
 ↪).
pattern_size = (9,6)

# Defining the world coordinates for 3D points
pattern_points = np.zeros((np.prod(pattern_size), 3), np.float32)
pattern_points[:, :2] = np.indices(pattern_size).T.reshape(-1, 2)
pattern_points *= 1

#### a)
# Find feature points with checkerboard images.
chessboards = [findCorners(filename, pattern_size) for filename in img_names]
```

```
for corners in [chessboard for chessboard in chessboards if chessboard is not␣
 ↪None]:
    img_points.append(corners)
    obj_points.append(pattern_points)

#### b)
# Get the camera matrix and the distortion coeffiecients (radial & tangential).
img_shape = cv2.imread(img_names[0], cv2.IMREAD_GRAYSCALE).shape[:2]
camera_matrix, dist_coefs = calibrateTheCamera(obj_points, img_points,␣
 ↪img_shape)

#### c)
# Undistort the images and plot them.
for filename in img_names:
    undistortImage(filename, camera_matrix, dist_coefs)
```


0
100
200
300
400
0
100
200
300
400
500
600
PRINTER

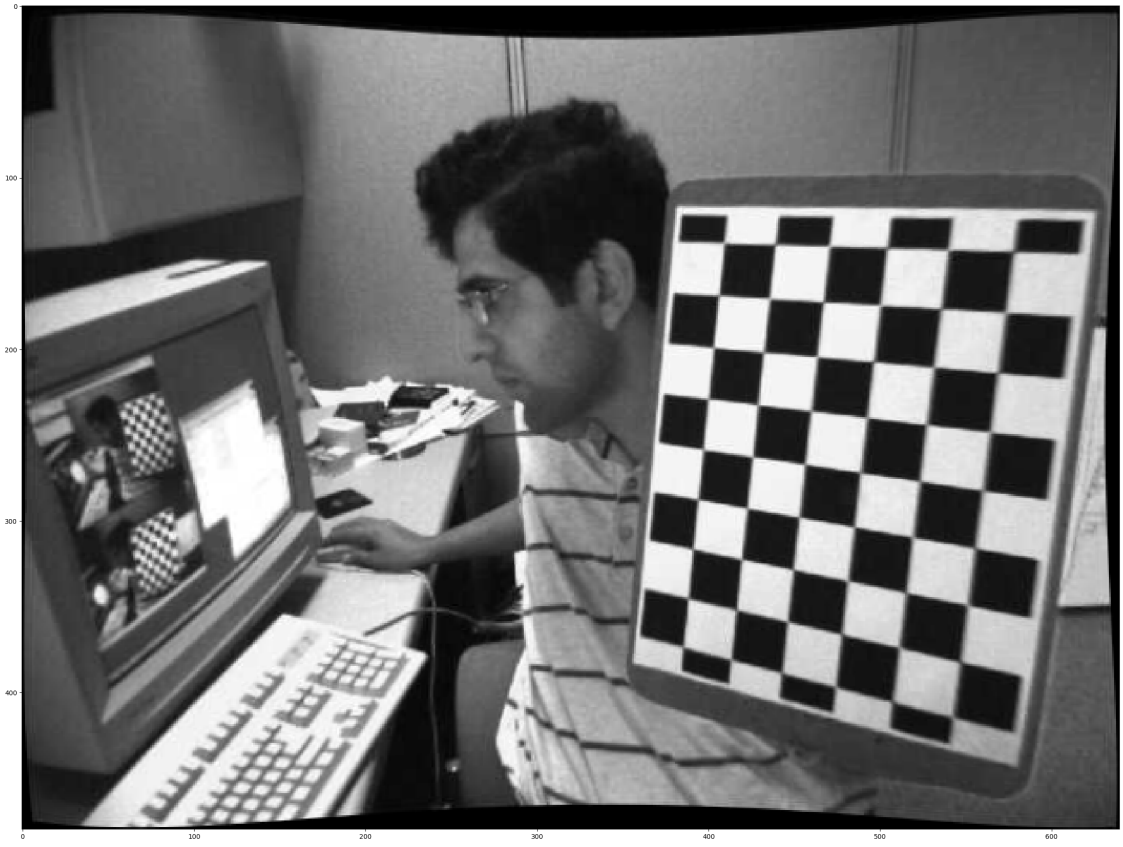

0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600
PRINTER


0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600

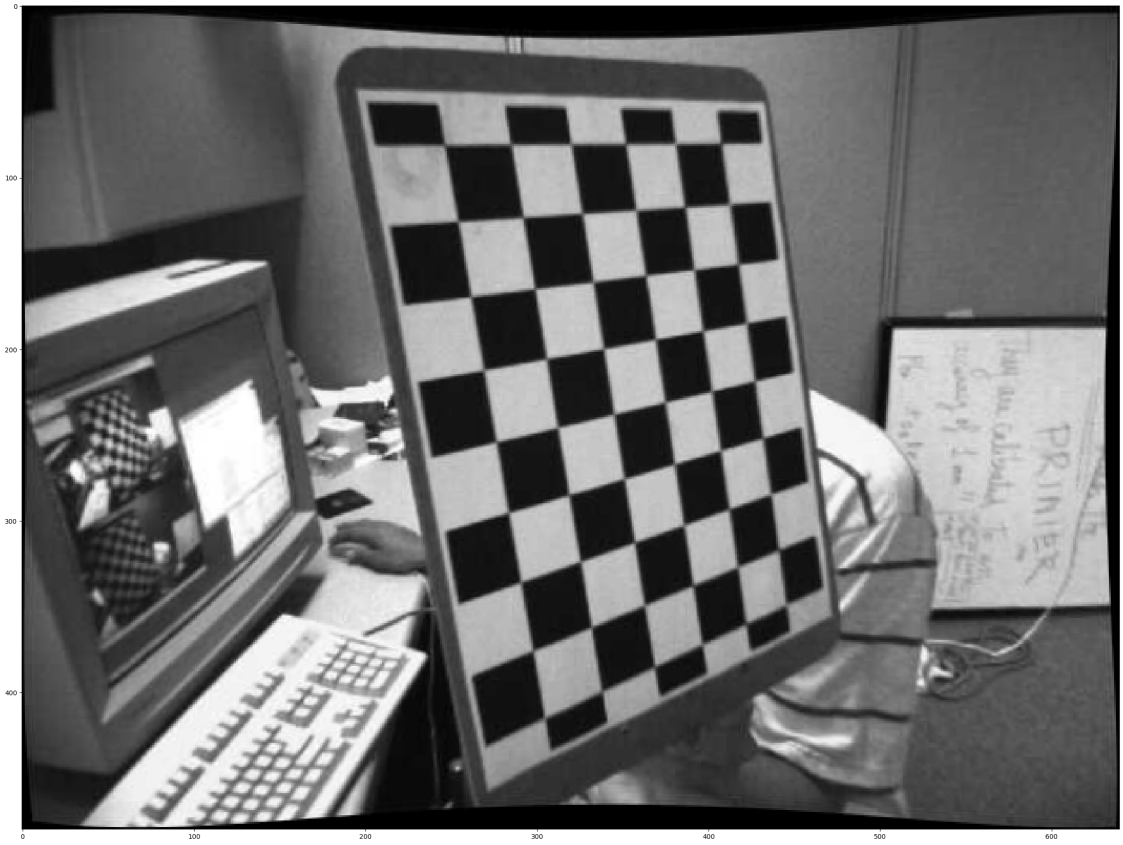

0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600
PRINTER


0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600
PRINTER


#### 1.3.1 Delivery (dead line) on CANVAS: 21.10.2022 at 23:59

### 1.4 Contact

#### 1.4.1 Course teacher

Professor Kjersti Engan, room E-431, E-mail: kjersti.engan@uis.no

#### 1.4.2 Teaching assistant

Saul Fuster Navarro, room E-401 E-mail: saul.fusternavarro@uis.no

Jorge Garcia Torres Fernandez, room E-401 E-mail: jorge.garcia-torres@uis.no

### 1.5 References

[1] S. Birchfeld, Image Processing and Analysis. Cengage Learning, 2016.

[2] I. Austvoll, "Machine/robot vision part I," University of Stavanger, 2018. Compendium, CANVAS.