# 1 Task Camera Calibratoin

**Perform a camera calibration.**

The images required for the calibration are located in the folder "exercise 1/images". The image you want to calibrate is a live image from the puzzlebot's camera. To retrieve it please use the utility program `getpicture` (utility-getpicture ). This program saves the camera image as `image.png`

```
#!/usr/bin/env python3

from opcua import ua, uamethod, Client

user = "puzzlebot"
password = "password"
CREDENTIALS = "{}:{}".format(user, password)
hostname = "engine.ie.technikum-wien.at"
port = 4840
CLIENT = Client("opc.tcp://{}@{}:{}/".format(CREDENTIALS, hostname, port))
try:
    CLIENT.connect()
    NODE = CLIENT.get_node("ns=4;i=324")
    frame = NODE.get_value()
    f = open("image.png", "wb")
    f.write(frame)
    f.close()
finally:
    CLIENT.disconnect()
```

Code 1: utility-getpicture

With the now available images you can calibrate the camera and display an image without distortion. First, some variables must be created.

- Variable for the edge length of a chessboard tile (15mm)
- An array for pixels found in the images
- An array for object points found in the images
- A Float32 matrix for all object points contained on an image (9x8x0)

*Note:* Create the matrix with the function `numpy.zeros((9*8,3),numpy.float32)`

Fill the matrix with the function `numpy.mgrid()` and cut it with `*.T.reshape()` Then multiply the

matrix by the tile length of the chessboard to obtain the correct shape of the chessboard as a virtual coordinate system.

*Note:*. If you are not too familiar with Python slicing, use the following line: `Matrix[:,:2] = numpy.mgrid[0:9,0:8].T.reshape(-1,2)* tile-length`

Now you can use the function `glob.glob("Folder/*jpg")` to define a range in which the images for the calibration are contained. The next step is to read in each of the images with a loop construction, step by step, and check them against a checkerboard pattern. With the function `cv2.imread("image.jpg")` an image can be read in.

Since the images can only be processed in grayscale, the image must be converted to the grayscale color space. Convert this with the function `cv2.cvtColor(image,parameter)`, where the parameter for grayscale is `cv2.COLOR_BGR2GRAY` With the function `cv2.findChessboardCorners(image,(DimensionX,DimensionY),parameters + ...)` you will find a chessboard with a certain dimension, if one exists in the respective image. For this exercise you use a chessboard of the dimension (9x8). For the parameters that this function takes, it is recommended to use `cv2.CALIB_CB_ADAPTIVE_THRESH` and `cv2.CALIB_CB_NORMALIZE_IMAGE` If the checkerboard pattern recognition process takes a long time the parameter `cv2.CALIB_CB_FAST_CHECK` can also be used. Please note that the function returns two values. The first value is of type Boolean and indicates whether the function was successful or not. The second value returns the found corners if the first value is true.

For the next step the corners found in the chessboard are recognized more accurately. For this purpose it must be checked whether a chessboard was found in the first step. For images where this is the case, you can use the function `cv2.cornerSubPix(image, found corners, size of the search window; e.g.: (5,5), size of the dead zone; e.g: (-1,-1)= no dead zone, parameter for abort criterion)` the corners are refined. The parameters for the abort criterion of this function are `cv2.TERM_CRITERIA_EPS` and `cv2.TERM_CRITERIA_COUNT` These parameters describe once the desired accuracy of the calculation of the corner, as well as the number of calculations, after which the process is to be aborted.

Now you have found the pixels for the object points. Now you have to add them to the array for pixels created at the beginning. To do this use the function `array.append(found corners)` For each valid image you must also add a matrix of object points to the array for object points. So for each image there is an entry in the array for image points and object points.

As a next step you could optionally have the detection of edges visualized. For this purpose the function `cv2.drawChessboardCorners(image, (DimensionX,DimensionY), found corners, True/False)` This function specifies an image with marked corners, as shown in figure 1.
With the function `cv2.imshow("window name",image)` an image can be displayed in a separate window. The window remains open as long as you define it in the function

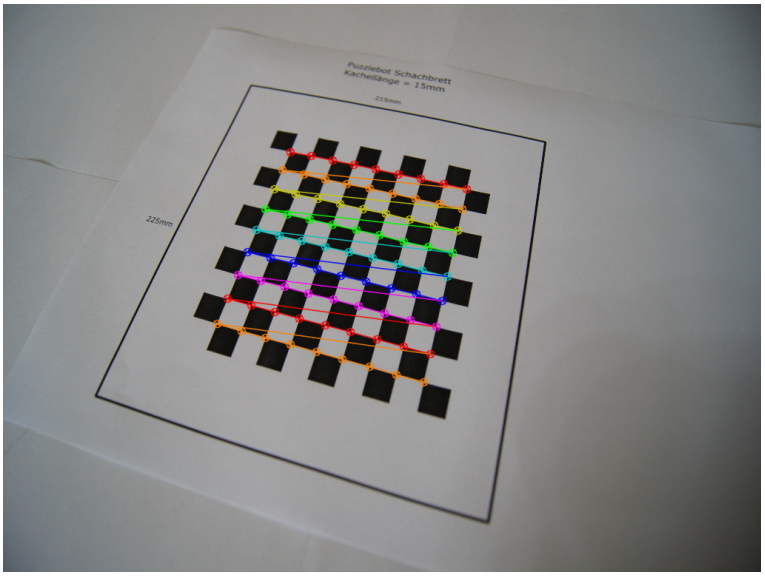


Figure 1: Found corners on the chessboard

`cv2.waitKey(time in milliseconds)` Finally, close the opened windows with the function `cv2.destroyAllWindows()`.

This gives you all the values you need to perform a camera calibration now. This calibration is performed with the function `cv2.calibrateCamera(object dot array, image dot array, resolution of images,parameters,abort criterion)` The resolution of the images can be specified as a value if known or can be retrieved with the function `image.shape[::-1]` Various parameters can be specified, but you do not need them for this exercise. Further information is available in the OpenCV documentation to find. You also do not need an abort criterion, since you only have a limited number of images available. For both values, please enter `None` in the function The camera calibration function returns a total of five values. The Boolean is important for you to know whether the function was successful or not. Equally essential are the camera matrix and the distortion coefficients. The last two values are vectors, which approximate rotatory and translatory movements of the images, and can be neglected here.

With the now existing camera matrix and distortion coefficients, new images can be equalized and displayed correctly. The calibration images are in a resolution of `1920x1080` pixels. In contrast, the live images from the Puzzlebot can only be displayed in a resolution of `640x480` pixel resolution. However, the camera matrix is intended for the higher resolution and must be scaled first, as can be seen in Formula 1.

$$\text{new camera matrix} = \text{scaling} \cdot \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{1}$$

Important here is that the camera matrix in Python is represented in an array, which can easily

be multiplied by a factor. After the multiplication you have to replace element [2][2] of the matrix with a one again, because the last row must not be affected by the multiplication.

Scale the matrix from `1920` Pixel by a factor of 3 down to `640` pixels. To correct the aspect ratio, scale the parameter $c_y$ up by 1,333.

After scaling the matrix, the camera image of the puzzle bot is now de-interlaced. With the function `cv2.getOptimalNewCameraMatrix(Camera Matrix,Distortion Coefficients,(Image Width, Image Height),Parameters,Center of Principal Point)` the Camera Matrix will be adjusted even better to the image. The function returns a new optimal camera matrix. The parameter of the function is a free parameter and can be chosen arbitrarily between 0 and 1. If the parameter is set to zero, the function cuts off unwanted pixels at the edge. These are caused by the distortion.

With the function `cv2.undistort(Image, Camera Matrix, Distortion Coefficients, None, new optimal camera matrix)` the image is now deskewed according to the camera calibration.

Use the libraries and function listed in chapter 1.1 to program task 1.

## 1.1 Functions and Libraries Exercise 1

Libraries:

- `import cv2`
- `import numpy`
- `import glob`

Functions:

- `glob.glob()` Create path for images
- `numpy.zeros()` Create an array filled with the number 0
- `cv2.imread()` Read an image
- `cv2.imshow()` Show a picture in a window
- `cv2.waitKey()` Display window of the image x milliseconds
- `cv2.destroyAllWindows()` Close window of the image
- `cv2.resize()` Resize image in height and width
- `cv2.cvtColor()` Convert the image to another color space

- `cv2.findChessboardCorners()` find corners of the chessboard
- `cv2.cornerSubPix()` Refine found corners
- `cv2.drawChessboardCorners()` visualize found corners on the chessboard
- `cv2.calibrateCamera()` Calibrate Camera
- `cv2.getOptimalNewCameraMatrix()` Customize Camera Matrix
- `cv2.undistort()` Destruct image