[ELE00142M] Machine Vision and Human Machine Interaction

Lab 2: Localization and Tracking of Dynamic Objects

Part 1: Calibration

Lecturer: Mark A Post <mark.post@york.ac.uk>

Demonstrator: John Bateman <john.bateman@york.ac.uk>

Technician: Alejandro Pascual San Roman <alejandro.pascualsanroman@york.ac.uk>

Aims and Objectives

In this lab session you will be experimenting with the Open Source Computer Vision Library (OpenCV) using a USB camera on a laboratory desktop workstation. The focus of this lab will be the use of a calibrated camera to localize and track features in 3-D space.

It is expected that you will complete all of these activities in two weeks of labs (Week 4 - Week 5). The P/T/410 and P/T/411 laboratories are open for your use outside of scheduled lab times in case you need extra time to finish. You can also work at home by installing the Python language and OpenCV-Python on your own computer.

Learning outcomes

- Repeated image capture, processing and saving in OpenCV
- Understanding of camera coordinate systems and calibration
- Calibration of a camera using a calibration pattern
- Undistortion of images given camera parameters

Hardware and Software required

- Python 3 with OpenCV-Python and numpy libraries installed
- Logitech C930 or similar USB camera with autofocus
- Calibration image files *pattern_chessboard.png* and *pattern_acircles.png* provided on the module webpage. They are also provided at the end of this document.

Tasks

1. Program a Digital Camera

To perform camera calibration, and facilitate capturing multiple images that will be used later in the lab, a simple Python program outline can be used. As this program is a bit more complex than previous examples it is provided all together, with comments (after '#' symbols) to explain its operation, and serves as an example of Python language features you can use in your programs. Create a new Python script text file and save it to have extension '.py'. Then run it either in your IDE or from the command line with "python3 <filename>.py". Copy the following program code into it and save it.

```
import cv2
import numpy as np

#Create a VideoCapture instance
cap = cv2.VideoCapture(4)
#Check if the device could be opened and exit if not
if not cap.isOpened():
    print("Cannot open camera")
    exit()

#Set the resolution for image capture
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

#Set up windows for the video stream and captured images
cv2.namedWindow("Video Stream")
cv2.namedWindow("Captured Image")
```

```
#Initialize an empty list of images and the number to be captured
number_of_images = 10
imglist = []
success = True

#Loop through the indices of images to be captured
for imgnum in range(number_of_images):

    #Capture images continuously and wait for a keypress
    while success and cv2.waitKey(1) == -1:

        #Read an image from the VideoCapture instance
        success, img = cap.read()

        #Display the image
        cv2.imshow("Video Stream", img)

    #When we exit the capture loop we save the last image and repeat
    imglist.append(img)
    cv2.imshow("Captured Image", img)
    print("Image Captured")

#The image index loop ends when number_of_images have been captured
print("Captured", len(imglist), "images")

#Save all images to image files for later use
for imgnum, img in enumerate(imglist):
    cv2.imwrite("Image%03d.png" % (imgnum), img)

#Clean up the viewing window and release the VideoCapture instance
cv2.destroyWindow("Captured Image")
cv2.destroyWindow("Video Stream")

cap.release()
```

The function of this program code should be clear from the comments - and program code should always be well commented to describe its function.

Note that if you have more than one camera connected to your computer, you may need to pass a different number to *cv2.VideoCapture()*. For example, in Linux and most UNIX-like operating systems *cv2.VideoCapture(0)* will open the camera device at */dev/video0*, *cv2.VideoCapture(1)* will open the camera device at */dev/video1*, and so on. Make sure you know which enumerated device is the camera that you want.

Run the program and make sure that you are capturing video from your camera in the "Video Stream" window and that you can save several images in the “Image Captured” window by pressing a key. The images are written to *.png* Portable Network Graphics files in the directory you run your code, make sure you can view them (you can change the extension and OpenCV will automatically write the file in a different format if you need, e.g. *.jpg*).

Now you see how easy it is to create a digital camera in Python! 🐍

2. Understand the Background of Camera Calibration

The purpose of a camera is to project as accurately as possible the light rays entering its lens (or a pinhole) onto a sensor element. However, due to the nonlinear nature of light refraction in physical lens elements, most cameras have a degree of *distortion*, particularly in the case of very wide angle or magnifying telephoto lenses or “pinhole” cameras that have no lens element. In addition, camera CCD and CMOS sensors operate on a very small projection area and do not produce a scale-accurate image without unit conversion calibration.

This lab focuses on using features to identify the position of objects in 3-D space, using only 2-D images. To obtain accurate reconstructions of the scenes that cameras view, it will become necessary to *calibrate* the camera by obtaining a matrix of values that, when applied to the image coordinates of features, will counteract the camera’s built-in distortion. It is advisable (though not absolutely mandatory) to calibrate the camera you are using so that image features can be transformed into scale-correct 3D coordinates. Note that each camera is likely to have a different calibration and as such must be calibrated separately if you use it. For details on the calibration procedure you can refer to the OpenCV Calibration Tutorial, and note that you can construct the calibration matrix (noted as “unit conversion” in the tutorial) yourself using information about the camera focal length, and resolution with pixel centres. A similar solution for camera calibration is detailed in this Camera Calibration using OpenCV tutorial.

We can calibrate for two kinds of distortion. Radial distortion is also known as "barrel" or "fish-eye" distortion, making images look "bent" in a circular pattern. It is modelled by transforming a point *(u,v)* on the image to distorted coordinates

$u_{radial} = u(1+k_1r^2+k_2r^4+k_3r^6)$
$v_{radial} = v(1+k_1r^2+k_2r^4+k_3r^6)$

Tangential distortion is caused by camera lenses that are not perfectly parallel to the image sensors behind them and is represented with distorted coordinates

$u_{tangential} = u+[2p_1uv+p_2(r^2+2u^2)]$
$v_{tangential}=v+[p_1(r^2+2v^2)+2p_2uv]$

If we take the characteristic coefficients from these equations that represent the distortion of all image points *(u,v)*, we can collect them in one vector of *distortion coefficients* as *distortion_coefficients=(* k_1, k_2, p_1, p_2, k_3 *)*.

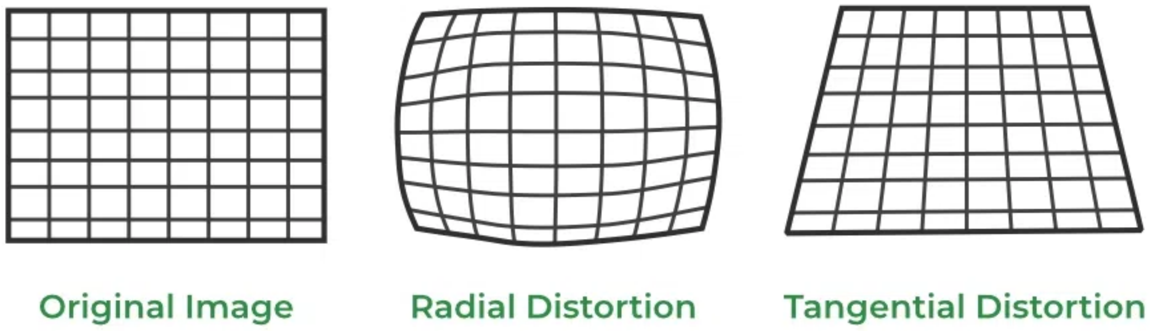

Figure 1: Effects of Image Distortion

Lastly we need to scale the image coordinates *(u,v)* to make them accurate to physical coordinates in three dimensions *(X, Y, Z)*. For this we need to consider the camera optical centre in pixel coordinates (c_u, c_v) and the camera focal lengths f_u in *u* and f_v in *v*, where generally $f_v = f_u * a$ with *a* as the aspect ratio of the camera (e.g. *4/3* or *16/9*). For convenience we use a *homogeneous matrix* (a matrix that combines multiple transformations)

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f_u & 0 & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

The term *w* may be confusing - it is of course a two-dimensional image! However it is just for mathematical convenience, and since the bottom row of the matrix is 0 except for a 1 in the bottom-right corner (one row of an identity matrix), then *w*=*Z*. Because these parameters are *intrinsic* to the camera (i.e. by its design and implementation), they are called *intrinsic parameters* and the matrix that transforms *X*, *Y*, and *Z* into *u* and *v* is called the *Intrinsic Matrix*.

The camera is assumed to be placed in a World Coordinate System (WCS), at a location defined by the Rotation and Translation of the camera in 3D space (you can measure it relative to a given location in space with a ruler and a protractor). These are *extrinsic* (externally-defined) to the camera and are generally placed together in another homogeneous matrix called the *Extrinsic Matrix*, usually defined by combining a rotation and translation matrix as $E = [\,R \mid T\,]$. The tutorial Geometry of Image Formation contains more details on this.

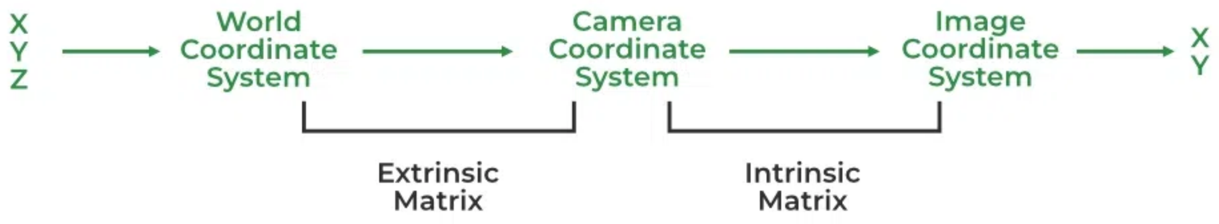

Figure 2: Coordinate Systems used in Machine Vision

The calibration problem now becomes: how to find these coefficients k_1, k_2, p_1, p_2, k_3, c_u, c_v, and f_u? As when solving for any unknown parameters, it is necessary to have an equal or larger number of equations that can be solved to obtain them. We can define these equations by using the models of camera distortion above and putting in sets of *(u,v)* image points that correspond to *points in 3D space with known relative locations*. These points *(X, Y, Z)* in 3-D space are within our “world” coordinate system. For this, we will use either the calibration image *pattern_chessboard.png* or *pattern_acircles.png*, printed out on paper sheets. These patterns are used because they have features of known size at regular and even spatial intervals, and are also easy to detect with keypoint detectors that we have already used.

3. Calibrate Your Camera

OpenCV includes several functions specifically designed to facilitate camera calibration for the provided image patterns. You will need to obtain the image keypoints for *all* the images you capture. The keypoint locations in each image will provide parameters for one equation to be used for solving the calibration problem. You can either start a new loop with `for img in imglist:` in which you can go through all images and get their keypoints, or just place the calibration code in your main capture loop.

Initialize two lists before you loop through the images. The list *objpoints* will store the 3D points *(X, Y, Z)* in physical space and the list *imgpoints* will store the 2D points in the image plane *(u,v)*.

```
objpoints = []
imgpoints = []
```

If you are using a chessboard (checkered) pattern, you can use the *cv2.findChessboardCorners()* function - look up the OpenCV documentation for details of its arguments and flags. In particular you need to pass it the size of the chessboard in the number of rows and columns of *corner points used*, e.g. *(6,9)*. The *pattern_chessboard.png* image has 7 rows and 10 columns of squares, however the corner detection generally finds only *internal* corners i.e. the centre point of 4 squares, so we subtract one from both measures. We will use it by converting the image to greyscale and calling it for *each* input image captured with

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, corners = cv2.findChessboardCorners(gray, (6,9),
cv2.CALIB_CB_ADAPTIVE_THRESH + cv2.CALIB_CB_FAST_CHECK +
cv2.CALIB_CB_NORMALIZE_IMAGE)
```

To use these points for calibration, you will also have to define the *physically measured grid of points* on the checkerboard (i.e. the actual 3D layout of the chessboard corners or other keypoints to be found) such as (0,0,0), (1,0,0), (2,0,0) ....,(5,8,0). The following snippet of code can define this once at the beginning of your program because the pattern is fixed (and by comparing to the dimensions above you should be able to see how to resize it for other patterns). Needless to say you should keep it on a flat surface while you are calibrating!

```
objp = np.zeros((9*6,3), np.float32)
```

```
objp[:,:2] = 2.54*np.mgrid[0:6,0:9].T.reshape(-1,2)
```

Print out the array *objp* and check (with a ruler) that it describes the grid that you are using for your calibration pattern. The coefficient *2.54* is used because the printed grid you are using usually has 1 inch squares (much OpenCV development is done in the USA…). You should change this coefficient if you are using a pattern with different spacing or “pitch” between features.

You should be saving both the static object points and image points that you have found at each step of your program to use in final calibration, but *only if the image points are successfully found*. Use the return value from *cv2.findChessboardCorners()* for this

```
if ret == True:
    objpoints.append(objp)
    imgpoints.append(corners)
```

For convenient viewing, there is the *cv2.drawChessboardCorners()* function that annotates the image with the corners that have been found in the image, which can be called (this will write the annotations back to *img*) with

```
cv2.drawChessboardCorners(img, (6,9), corners, ret)
```

Include all of these elements in your program and try capturing 8 images of the chessboard image from different angles. Can you get the algorithm to correctly identify the center points of the chessboard in each one? You will need to hold the camera very steady to avoid blurring. You may find it useful to put the annotation function before the *imshow(“Video Stream”, img)* call so you can see in real time whether points are detected. Using the *cv2.CALIB_CB_FAST_CHECK* flag will exit the feature detection early if no keypoint images are found and speeds up video streams significantly.

Recommended: While the corner detection is fairly accurate, good camera calibration requires extremely precise (sub-pixel level) localization of 3D points, and an even more precise estimate of keypoint locations is desirable. OpenCV's *cv2.cornerSubPix()* function takes each corner location and searches the original image for the best corner location within a small neighborhood of the original location. You need to specify in order of parameters:

1. the input image
2. the initial coordinates of input corners to be refined
3. half the side length of the search window in pixels
4. half of the size of a deadzone in the middle of the search window, used to avoid possibly singularities, or *(-1,-1)* for no deadzone
5. Criteria for termination of the search, as a tuple of (desired accuracy *epsilon*, *maxCount* operations, and termination criteria)

One example of parameters that should work is

```
subcorners = cv2.cornerSubPix(gray, corners, (11,11),
(-1,-1), (cv2.TERM_CRITERIA_EPS +
cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001))
```

You will then need to append *subcorners* instead of *corners* to imgpoints.

Print some of the array elements of *corners[]* and print the same indexed elements from *subcorners[]*. Are they different? By how much has subpixel searching refined them?

4. Obtain the Calibration Parameters

Now that you have the object points *(X, Y, Z)*, and image points *(u,v)*, we can solve the equations necessary to obtain the camera parameters. To solve all the equations you need at least a predetermined number of pattern snapshots to form a well-posed equation system. For example, in theory the chessboard pattern requires at least two snapshots. However, in practice there is often a lot of noise present in our input images, so for good results you will probably need at least 10 good snapshots of the input pattern in different positions. This makes a large linear system of equations to be solved!

Fortunately, OpenCV provides the [cv2.calibrateCamera()]() function which does the complex work of obtaining the camera parameters in the form of:

- the camera intrinsic matrix *mtx*
- the distortion coefficients *dist*
- the rotation vectors for each image pose *rvecs*
- the translation vectors for each image pose *tvecs*

You need to pass it your object points, image points, the input image size, and optionally, flags and a tuple of termination criteria (the defaults usually work OK) Include the function in your code only after all object points and image points have been obtained, and possibly after closing the windows and camera since it will otherwise keep them open while it is solving the equations.

```
ret, mtx, dist, rvecs, tvecs =
cv2.calibrateCamera(objpoints, imgpoints,
gray.shape[::-1], None, None)
```

Note that the image size must be 2D (e.g. if you use img.shape[::-1] you will get (720, 1280, 3) which will confuse the function). You can also just pass in *(720, 1280)* as object size if you do not change the resolution of the image.

You can print the resulting matrices with

```
print("Intrinsic Matrix: \n")
print(mtx)
print("Distortion Coefficients: \n")
print(dist)
print("Rotation Vectors: \n")
print(rvecs)
print("Translation Vectors: \n")
print(tvecs)
```

Be prepared for the function to take quite some time to complete - the algorithm must find a set of solutions for all of the input images you have used. If you are having difficulty getting repeatable or good-looking results, try capturing more images, or decreasing the resolution.

You can try setting a tuple as the last argument to the function with different

Termination Criteria as you did in *cv2.cornerSubPix()*. The last element of the tuple of termination criteria is *epsilon* which is the desired accuracy of the calibration. If it is very small the solution may take a very long time! If it is larger the solution may finish faster but will not be as precise. Experiment with different values to find a good balance.

note that the default is:

```
cv2.calibrateCamera(…, …, …, 0
(cv2.TERM_CRITERIA_COUNT+cv2.TERM_CRITERIA_EPS, 30, DBL_EPSILON))
```

where DBL_EPSILON is the numeric limit for the data type - for IEEE 64-bit floating point it is on the order of 2.2204460492503131e-16! (needless to say you can go larger than this and sacrifice relatively little precision). You could try for example a tuple of

```
cv2.calibrateCamera(…, …, …, 0
(cv2.TERM_CRITERIA_COUNT+cv2.TERM_CRITERIA_EPS, 30, 0.0001))
```

which is likely to lead to a much faster (but less precise) result!

You could also try the Asymmetric Grid of Circles calibration pattern (see Optional Activity 1) which sometimes requires less images for a good solution.

When you do get a solution, **make sure to save all the calibration parameters/matrices you get** into a text file, Comma-Separated Value (CSV) file, or just by writing them down as you will need to use them for future activities. You can for example write a whole matrix to a CSV file with

```
with open('intrinsic_matrix.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(mtx)
```

Note: If capturing repeated images is tiresome or too time-consuming, you can just use the example code given save the original (un-annotated) images at the locations when a good set of keypoints is found to *.png* files (recommended as *.png* is lossless while *.jpg* is a lossy format). Then you can read in these files in a separate Python program and perform only the keypoint detection and calibration without the VideoCapture operations.

You can also look up and try the *cv2.calibrateCameraRO()* function instead which is designed to handle inaccurate calibration surfaces but requires estimated distortion coefficients.

5. Undistort the Images from your Camera

Finally, after all of this detailed work, you can obtain calibrated, dimensionally-correct image transformations from your camera. OpenCV comes with two methods for doing this. However first, we can refine the camera matrix based on a free scaling parameter using *cv2.getOptimalNewCameraMatrix()*. If the scaling parameter *alpha=0*, it returns undistorted image with minimum unwanted pixels. So it may even remove some pixels at image corners. If *alpha*=*1*, all pixels are retained with some extra black images. This function also returns an image ROI which can be used to crop the result.

```
h, w = img.shape[:2]
newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx,
dist, (w,h), 1, (w,h))
```

Using *cv.undistort()* is the easiest way. Just call the function and use ROI obtained above to crop the result.

```
dst = cv.undistort(img, mtx, dist, None, newcameramtx)
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]
cv.imwrite('calibresult.png', dst)
```

Alternately, using remapping is a little bit more difficult. First, find a mapping function from the distorted image to the undistorted image. Then use the remap function.

```
mapx, mapy = cv.initUndistortRectifyMap(mtx, dist, None,
newcameramtx, (w,h), 5)
dst = cv.remap(img, mapx, mapy, cv.INTER_LINEAR)
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]
cv.imwrite('calibresult.png', dst)
```

Both should give about the same results. Try comparing these results, though, particularly with images of the chessboard that have straight lines in them. Can you see a difference between the distorted and undistorted images?

OPTIONAL ACTIVITY 1:

We have used a checkerboard pattern for calibration throughout this lab, but in fact any evenly spaced, measurable, and reliably detectable pattern could be used for calibration (so long as the physical grid is known and matched with the image keypoints). There are a variety of well supported calibration patterns in OpenCV with keypoint detection functions such as:

- Chessboard, detected with *cv2.findChessboardCorners()*
- ChArUco (a chessboard with ArUco tags within the squares), detected with *cv2.aruco.CharucoDetector.detectBoard()*
- Grid of Circles, detected with *cv2.findCirclesGrid()*
- Asymmetric Grid of Circles, detected with *cv2.findCirclesGrid(..., … , …, CALIB_CB_ASYMMETRIC_GRID)*

You have been provided with an Asymmetric Grid of Circles in *pattern_acircles.png*. Copy your working calibration program to a new Python source file, and using the same process as you have seen already, try replacing the Chessboard functions in your calibration program with the functions to detect and calibrate the camera using the Asymmetric Grid of Circles instead. Refer to the OpenCV documentation and search for tutorials on the Internet as needed to fill in gaps in your knowledge.

Compare the camera parameters you get with the two different calibration programs. Does one work better than the other? More reliably? Can you achieve a good calibration with less snapshots of the circle patterns than were needed for the chessboard? Consider why and how different calibration patterns could be used for different calibration requirements and scenarios.

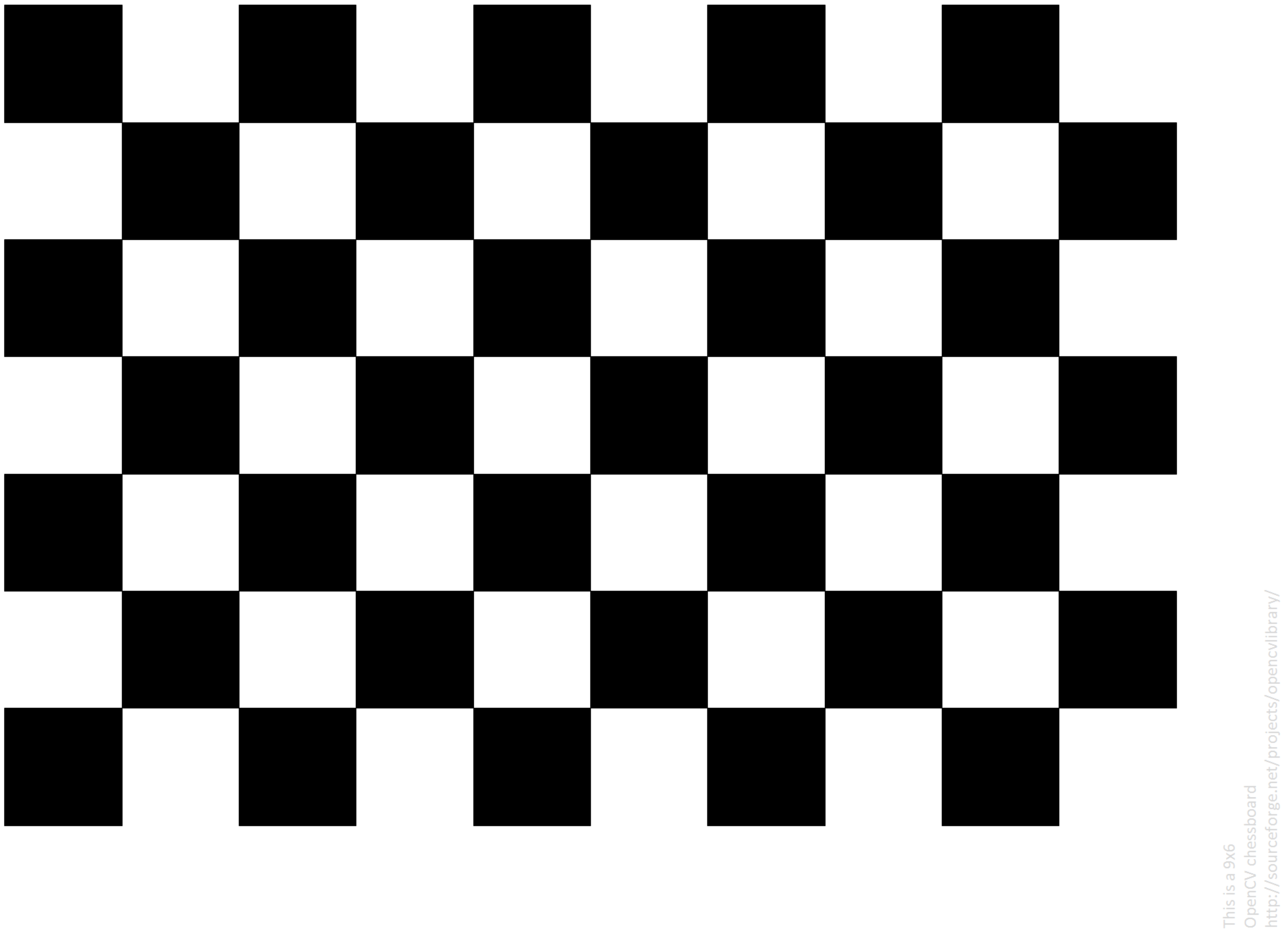

Figure 3: Chessboard Calibration Pattern (pattern_chessboard.png)

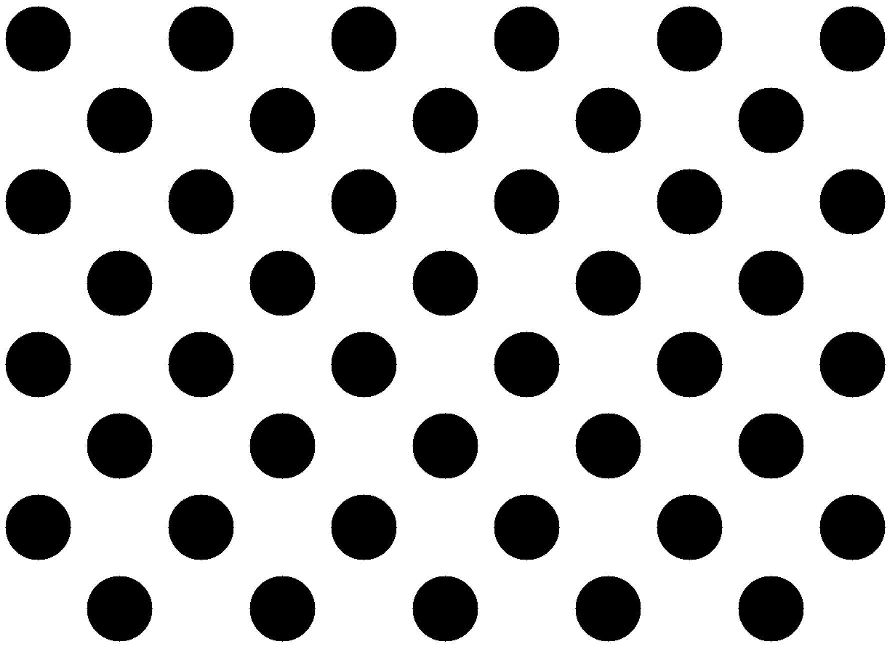

Figure 4: Circles Calibration Pattern (pattern_acircles.png)