ISTANBUL TECHNICAL UNIVERSITY
Faculty of Computer Science and Informatics

Estimation of Angle for Actuator Bending using Image Processing

INTERNSHIP PROGRAM REPORT

Denıs Iurıe Davıdoglu
150200916

Summer / 2023

Table of Contents

1 INFORMATION ABOUT THE INSTITUTION

Soft Sensors Laboratory is located in the Ayazağa campus of Istanbul Technical University, founded in 2018 by Dr. Özgür Atalay, Dr. Gökhan İnce and Dr. Aslı Atalay. The laboratory was intended to be a place for research in the multidisciplinary field of wearable technology, which encompasses textile, electronics, and computer science. The institution received several grants from prominant organizations, including our university, European Union, Scientific and Technological Research Council of Turkey (TÜBİTAK) and Pakistan Science Foundation (PSF) [1].

According to their website [2], among the 19 people currently active, there are professors and students from Textile Engineering, Computer Engineering, and Control and Automation Engineering. The projects either finished or still worked on by them are [3]:

- Development of Textile Based Motion Capture System
- Fabric based Soft Actuators for Wearable Applications
- Textile based soft sensing actuators for soft robotic applications – TexRobots
- Development of Textile Based Robotics for Foot Drop Syndrome
- Sports Activities with Wearable Technologies
- Textile-Based Wearable Soft Robotics with Integrated Sensing, Actuating and Self Powering Properties – TEXWEAROTS
- ReTex – Regenerated Cotton for Electronic Textiles

The projects constantly generate findings worth sharing with the scientific community by means of publications, seminars, conferences, and collaborations with other institutions.

The internship was beneficial for me because it provided me with hands-on experience in advanced topics of object-oriented programming, computer vision and software engineering.

2 INTRODUCTION

My internship work was about developing an image processing algorithm for estimation of a textile actuator bending angle. The machine-knitted pneumatic actuator developed by the laboratory can act as a motion sensor, but it is also capable of converting signals back to motion by inflating the actuator under the right pressure [4]. In both cases, a relation between raw signals and meaningful angle values is necessary. Instead of manually measuring the angle of the bending actuator using a ruler, a better approach would be creating an automated computer vision system for estimating the angle based on the video frames. In less time, more data is collected, which can be fed to a machine learning model to correlate the angle values with signals. I used OpenCV library with Python bindings to create an application capable of estimating the angle, correcting the distortions induced by camera's perspective, recording, playing back and saving the data into a CSV file.

3 DESCRIPTION AND ANALYSIS OF THE INTERNSHIP PROJECT

3.1 OpenCV and OOP in Python

OpenCV stands for Open Source Computer Vision Library, available in many programming languages including Python. The library has thousands of implemenations of image processing and machine learning algorithms [5]. The users of the library focus on applying the algorithms instead of implementing them from scratch, which results in a more gradual learning curve in developing computer vision applications.

My journey with OpenCV started from learning the basics input/output functions. I learned how to read and write image files, convert them to raw bytes, capture video from a camera and draw windows. Then, I tried the basic image processing techniques such as applying low and high pass filters, edge detectors and shape detecting algorithms. For example, I made a picture of the textile actuator and performed several steps:

1. Load the original image (Figure 1a)
2. Blur and apply custom kernel, making the image more contrasted (Figure 1b)
3. Select the blue channel, which is the most pronounced in the actuator (Figure 1c)
4. Apply thresholds and find contours (Figure 1d)

As it is seen from the figures and their respective code (Code snippet 1), OpenCV library and Python are powerful in image processing, because of the high quality routines and ease of use. With just a little knowledge of computer vision, extracting the required features becomes trivial, and the programmer can focus on interpreting them. After writing this example, I thought I could take the contours, reduce the number of points and create a machine learning model to calculate the angle at the vertex. However, I had no experience in artificial intelligence at all, and a more suitable approach would be placing markers to track certain positions. Before explaining the two methods I developed, I would like to describe the architecture of my application.

Object-oriented programming was not the paradigm I initially thought of using. When learning OpenCV, I was writing short codes in separate files to see how each functionality worked. Once I began combining these small scripts, the code became unorganized and hard to understand. Also, with multiple ideas of how to implement the application, it is desirable for the general interface to be similar. I realized that I could encapsulate the entire application into a single class, with the following interface:

- `setCamera(source)`. Selects a camera source to be used as input. For example, `setCamera(0)` selects the first camera the computer connected to during the boot. Uses OpenCV's built-in routine `cv2.VideoCapture()`.
- `setInputFile(path)`. Selects a directory of a recording to be used as input. The directory must contain a CSV file and frame images. The function parses the path to find the "template" name, then opens the recording's CSV file in the read mode. For example, from "~/Videos/actuator/recording1" it sets the template name to be "recording1", and then finds and opens the "~/Videos/actuator/recording1/recording1.csv" file.
- `setOutputFile(path)`. Selects a destination directory for recording. If such a path already exists, it is overwritten. If the path does not exist, the function

automatically creates all specified subdirectories. The template name is selected as in the setInputFile(path) function's description, a CSV file is opened in the write mode, and the same template is saved to be used as the base for the frame names.

- run(). Used in the application's while loop to fetch and process frames, draw window, save data and respond to keyboard events. All of these are private methods.
- close(). Used in the end of program to close window, camera and files.
- output_mode is a boolean which indicates whether the recording is active and is disabled by default. It can be used to start and pause the recording when needed. When **R** key is pressed, this flag is toggled.
- success is a flag intended to be used as condition of the application's while loop. When camera gets disconnected or **Backspace** key is pressed, the flag becomes False.

I provided some examples of how I intended the above functions to be used. Code snippet 2 is an example of how to create an Application instance with the window title "Marker tracking"; set the camera "/dev/video2" as input, that is the secondary USB webcam; enable recording; set the output path for the recording; create the application loop; and close the application.

Code snippet 3 differs from the previous one in the fact that it uses an existing recording on the file system to perform processing. This is useful because the algorithm improvements can be observed by comparing the performances on the same input. In the while loop, there is an extra delay that prevents all the frames flashing at once, making it perceived as if it was happening in the real-time. In the Code snippet 2 this delay is naturally caused by the camera's frame rate.

However, if no extra features or events are desired in the program, there is no need to explicitly call the Application class functions. Instead, the program works out of the box with the following command line interface:

- -c or –camera. A camera source is selected as video input. The argument is either an index (e.g. 0, 1, 2, etc.) or a path to device (e.g. /dev/video0).
- -i or –input. A recorded file is selected as video input.
- -o or –output. Indicates path where the recording will be saved.
- -d or –delay. Useful when a recorded video is played and a slower frame rate is desired. Delay is specified in seconds, although the actual delay between the frames also varies depending on the image complexity.
- **Note:** –camera and –input cannot be specified together.

Private methods' names are almost similar across code versions, but naturally, their inner structure is different.

- __getFrame(). Depending on the input type, that is either live video capture or replaying a recording, new frame is obtained and stored inside the object instance.
- __findAngle(). The current frame is processed by finding markers, calculating their center points, correcting the points using perspective transformation, and finally calculating the angle using the law of cosines.

- __perspectiveCalibration(). Calculates the perspective transform matrix.
- __drawWindow(). All changes to the frame are done in this function. Before drawing anything, the frame is saved as "original" if the recording in on. Then, although computationally expensive and unnecessary, perspective tranformation is applied to the whole frame for the demonstration purposes. If all three markers of the actuator are found, they are marked with points and connected with lines. The frame is extended on the bottom to provide information about the angle and recording status.
- __keyboardResponse(). The application uses three keys. **Backspace** is for closing the window. **C** is for perspective calibration. **R** is for toggling the recording.

The next subsections explain about their work principles.

3.2 Colour-Based Detection

The first method I ended up implementing was using colour-based detection of markers. Essentially, my algorithm would look for a specific pixel colour value with a threshold, in this case the white of the marker. Using the technique demonstrated in the Subsection 3.1 with applying a custom kernel, blurring and channel selection, the image is simplified and contours can be found. The actuator will have three white square markers on it, so that the angle at the middle point can be calculated. In fact, there is a large range of colours available for the marker, because the image is processed in the blue channel. Figure 2 demonstrates how colors are transformed, with the colorful gradient serving as a deeper reference.

There are a few conditions to be checked in order to find a marker among dozens of contours. First, the marker is neither too large nor too small. The main object, the actuator, has the longest contour, while the unwanted tiny particles on the image have the shortest contour. Second, the marker's contours must surround an area with mostly white pixels. And third, its shape is a square.

These cases are satisfied in the Code snippet 4. The contours are sorted by their length, from longest to shortest (line 1). In this way, the potential square markers will be in priority. In the for loop, each contour is approximated with a polygon, allowing a distance error up to 10% between the original path and the approximation (lines 6-7). A mean value of all pixels inside the region is computed (lines 8-10). And finally, the if statement at line 11 checks if the calculated mean pixel value is above a certain threshold, meaning that the region is mostly white, and the number of line segments which constitute the approximation is checked to be 4, that is a square.

When the marker is detected, its center's x and y coordinates are computed using the cv2.moments() function, and saved for further manipulations (lines 12-15, Code snippet 4). To observe the correctness of my program, I display a window playing the video capture with some marks on top, added by software. Figure 3 proves clearly that the code functions mostly correct.

There is a catch, however. In the case when the actuator bends at a too narrow angle, making a closed contour, it is falsely regarded as a marker (Figure 4). To fix this issue, I introduced some overhead by keeping track of the previous positions of the markers and trying to match them to the closest new marker candidates. For example, I have a calibrated setup, with only 3 markers visible in the first frame. A foreign object appears in the second frame and becomes regarded as a marker too. Now that there are 4 markers, the distances between them and the previous frame's 3 markers are computed. By minimizing the distance cost, it is possible to ignore any number of foreign objects, as long as the camera's frame rate allows

precise tracking. There is always a possibility to manually reset the system while the application is running, if wrong points are being tracked.

There can be a serios error in the angle estimation due to the perspective distortions. I should account for the cases when the camera is not looking exactly perpendicularly to the actuator's plane, which is the most realistic scenario. A perspective transform can be applied; however, it is not trivial to properly compute the transformation matrix. The only information about the space is given by the marker, which can be cut into ideal squares. With the knowledge of the real shape and with the detected points of the distorted version, a correction matrix can be computed.

I implemented a version of the colour-based detection program involving this kind of computation, and despite it performing better on high perspective distortions on average, the variance of the computed angles drastically increased. In other words, in similar frames with angle not changing at all, the estimated angle values can vary, with the difference of around 7%. Figure 5 shows two frames coming one after another, interpreted completely different. Notice that the perspective transform is taken in respect to the middle square, which appears corrected, unlike the other two. It can be concluded that neither of the transformed frames are correct, since in a proper perspective transform all the markers would appear as perfect squares.

As mentioned earlier, a 10% error is allowed when approximating the contour of the square marker. Due to their small size, this method is unreliable. To get the perspective tranform working, I totally changed the code to use something more robust.

3.3 ArUco Marker Detection

In computer vision, a more common type of markers uses patterns that make it clearly distinguishable from the environment. Similar to black and white QR codes, they might encode some information but with a much lower density. The first step to detect such a marker is to look for a specific topology, either a square or a circle. In fact, looking for a square was part of the algorithm I implemented and discussed in the Subsection 3.2. Instead of just averaging the pixel values inside the square to detect a colour range, advanced techniques can read a binary code from the marker. This is the most robust method, completely immune to false detections, since a specific binary sequence cannot occur randomly in an image [6].

OpenCV library implements this technique, also know as binary square fiducial markers identification. Its ArUco marker dictionaries include more that one thousand distinct markers. In the Figure 6 below, I indicated the markers used in the new version of the application.

There are still three markers put on the actuator, but not all are of the same type. Marker in Figure 6b with the ID of 9 is placed at the vertex of the actuator, and is the point at which the angle is calculated. Next, there are two copies of the marker with the ID of 8 (Figure 6a), which are placed on the actuator's tips. The program detect these three points, connects them by lines and computes the angle, exactly the same as in the previous version, except that the vertex is now unambiguous. In addition, there are four copies of the marker having the ID of 42 (Figure 6c), which are arranged on the background in the shape of a square. Their purpose is to provide a spacial reference to the plane of the actuator, an information which, combined with the knowledge that the points on the background represent a square, is used to compute the perspective transform matrix.

In the Code snippet 5, OpenCV's marker detector object is demonstrated. After finding all markers in the image, markers with the ID 42 are selected in the for loop (lines 1-6). Later, to correct these points to appear as a square, a square boundary is formed based on the frame size, and a perspective transform matrix is calculated (lines 7-14). In similar way, the other markers can be detected, but before calculating the angle of the actuator, the points are multiplied with

the perspective transform matrix. Hence, the angle is precisely calculated even when the camera is extremely tilted.

Figure 7 is a screenshot of the application's final design. To observe the accuracy of the measurement, I printed a test sheet containing all markers and angle divisions varying by 15 degrees. One of the markers is free to move, so it is possible to check for any angle. I can firmly say that the system functions in a robust manner, having a minuscule error of not bigger than 1 degree. A demonstration video posted on internet proves this statement [7].

3.4 Recording Data

As a last note, I discuss the format in which I save the recorded data. Recording was essential in the development stage, because it enabled me to compare different algorithms' performaces; but more importantly, the application should export the data to other programs for further interpretation.

When the output file name is specified using the application interface, a directory with that name is created. Furthermore, every file inside that directory is named similarly, being followed by unique IDs and a format extension. One of the files is the CSV record file, an example of which is the Table 1. `xn` and `yn` are coordinates of the *nth* point, `vertex` is the index of the point where the angle is measured, `angle` field is for angle in degrees, `orig_path` is the path to the original frame captured by the camera, and `marked_path` is the path to the frame which was processed.

x1	y1	x2	y2	x3	y3	vertex	angle	orig_path	marked_path
531	222	314	250	380	214	2	21	./r/r_orig_1.png	./r/r_marked_1.png
555	215	332	236	401	202	2	20	./r/r_orig_1.png	./r/r_marked_2.png

Table 1: CSV example

The CSV file can contain a comment line like this: `# c pressed`. This means that the **C** key was pressed, making the program calculate a new perspective transform. This is useful in replaying recordings.

4 CONCLUSIONS

Over the course of one month, I learned the basics of computer vision from scratch and was able to develop a useful program. I read many research papers on the techniques and algorithms for object detection, consulted computer vision textbooks and video lectures. I used the Python programming language to design a scalable software, while maintaining compatibility between different versions' interfaces and recorded data.

I compared the colour-based detection with binary square fiducial marker detection, drawing the conclusion that the first method has the advantage of speed, and the second one is more robust. Although requiring more overhead to decipher the marker's code, the ArUco marker version of the code, in my opinion, is better, because it supports tracking multiple types of markers and has better angle estimation.

The Soft Sensors laboratory can afford special tools, machines, textile samples and has a valuable human resource. I was assisted throughout the internship, taking great advices and feedback, which enormously helped me in developing the application in a short period of time. From the engineering and academic points of view, the institution is very successful, and I would recommend it to anyone wishing to do research in image processing, machine learning, embedded systems and textile.

5 REFERENCES

[1] Soft Sensors Lab, *About Us*, https://softsensorslab.itu.edu.tr/about-us

[2] Soft Sensors Lab, *People*, https://softsensorslab.itu.edu.tr/people

[3] Soft Sensors Lab, *Projects*, https://softsensorslab.itu.edu.tr/projects

[4] Elmoughni HM, Yilmaz AF, Ozlem K, Khalilbayli F, Cappello L, Tuncay Atalay A, Ince G, Atalay O. *Machine-Knitted Seamless Pneumatic Actuators for Soft Robotics: Design, Fabrication, and Characterization.* Actuators. 2021; 10(5):94.

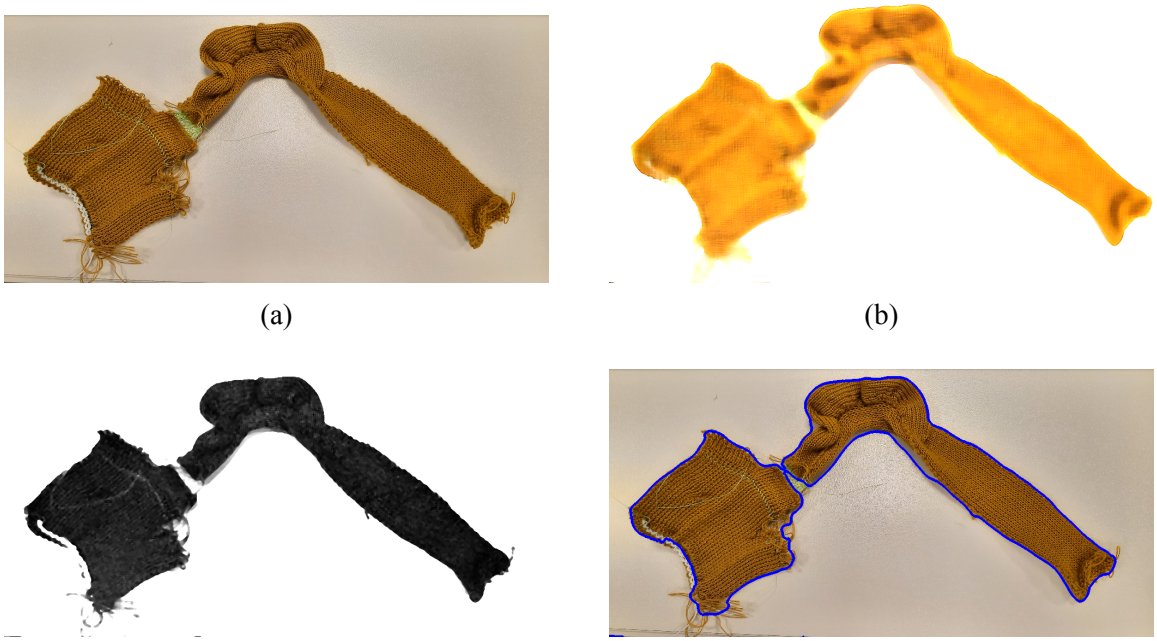
[5] OpenCV, *About*, https://opencv.org/about/

[6] J. Köhler, A. Pagani, D. Stricker. *Detection and Identification Techniques for Markers Used in Computer Vision*. 2010.

[7] Gökhan İnce, *Angle estimation using two markers*. https://youtu.be/YJUO5Wk2Srk?si=xR4gpN_dWDaI12lG

6 APPENDIX

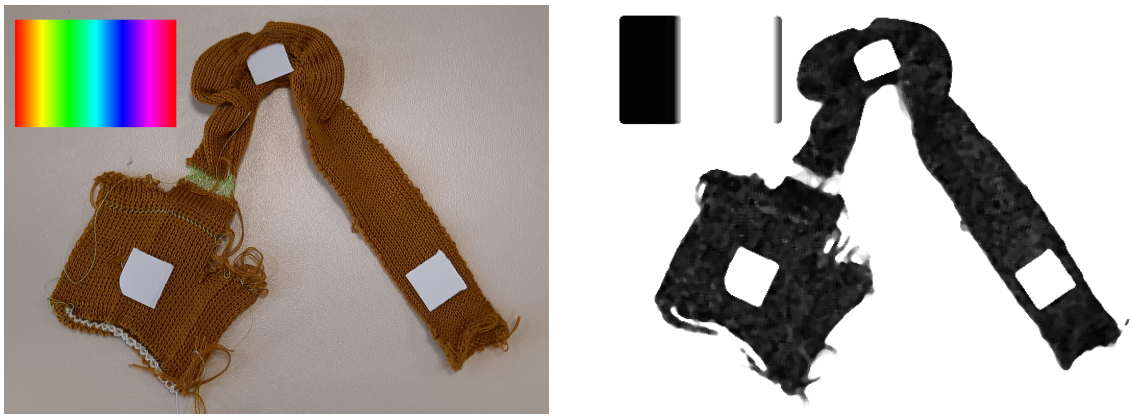
6.1 Figures



(a) (b)

(c) (d)

Figure 1: Contour detection steps



(a) (b)

Figure 2: Original image and processed image with affected colours

Figure 3: All points detected and marked with red dots

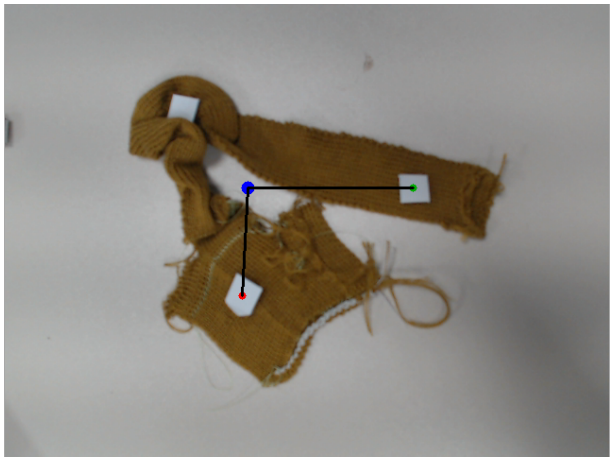


Figure 4: False detection of the vertex

(a) (b)

Figure 5: Almost identical frames get transformed completely different

(a) Size 4x4, ID 8 (b) Size 4x4, ID 9 (c) Size 4x4, ID 42

Figure 6: ArUco markers in use

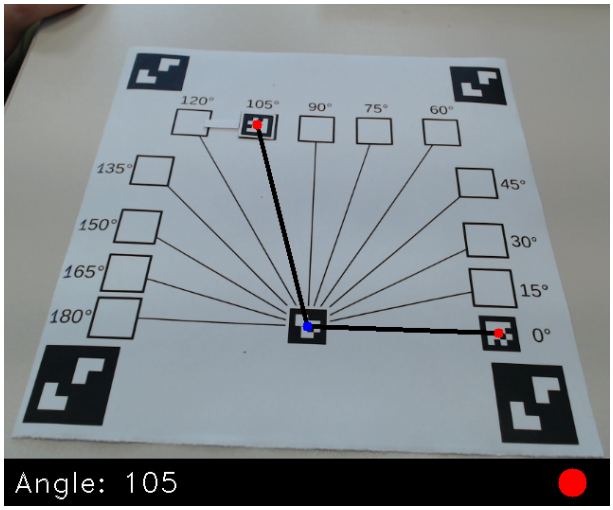


Figure 7: Application's final look

6.2 Code snippets

```
img_orig = cv2.imread(IMG_PATH, cv2.IMREAD_REDUCED_COLOR_2)
BLUR_FACTOR_1 = 13
BLUR_FACTOR_2 = 3
kernel = numpy.array([[-1,  -1, -1],
                      [-1,  11, -1],
                      [-1,  -1, -1]])
img = cv2.medianBlur(img_orig, BLUR_FACTOR_1)
img = cv2.filter2D(img, -1, kernel)
img = cv2.medianBlur(img, BLUR_FACTOR_2)
img = img[:,:,0]
thresh = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY_INV |
                                    cv2.THRESH_OTSU)[1]
contours = cv2.findContours(thresh, cv2.RETR_TREE,
                                    cv2.CHAIN_APPROX_SIMPLE)[0]
cv2.drawContours(img_orig, contours, -1, (255, 0, 0), 3)
```

Code snippet 1: Contour detection

```
app = Application("Marker tracking")
app.setCamera("/dev/video2")
app.output_mode = True
app.setOutputFile("./recordings/test")
while app.success:
    app.run()
app.close()
```

Code snippet 2: Example of using camera and recording via my interface

```
app = Application("Marker tracking")
app.setInputFile("./recordings/test")
while app.success:
    app.run()
    time.sleep(0.1)
app.close()
```

Code snippet 3: Example of replaying a recording via my interface

```
contours = sorted(contours, key = lambda x: cv2.arcLength(x, True),
                  reverse=True)
points = []
frame_copy = cv2.cvtColor(frame_copy, cv2.COLOR_GRAY2BGR)
for c in contours:
    epsilon = 0.1 * cv2.arcLength(c, True)
    approx = cv2.approxPolyDP(c, epsilon, True)
    mask = numpy.zeros_like(frame_copy[:, :, 0])
    cv2.drawContours(mask, [approx], -1, 255, -1)
    mean = cv2.mean(frame_copy, mask=mask)
    if (mean[0] > Application.MARKER_THRESHOLD) and len(approx) == 4:
        m = cv2.moments(approx)
        x = int(m["m10"] / m["m00"])
        y = int(m["m01"] / m["m00"])
        points.append([x, y])
```

Code snippet 4: White square marker search

```
markerCorners, markerIds, _ = Application.detector.detectMarkers(self.frame)
markers = []
for i in range(len(markerCorners)):
    if markerIds[i][0] == 42:
        center = sum(markerCorners[i][0])/4
        markers.append(center)
PADDING = 50
boundaries = numpy.float32([
    [PADDING, PADDING],
    [self.frame.shape[1]-PADDING, PADDING],
    [self.frame.shape[1]-PADDING, self.frame.shape[1]-PADDING],
    [PADDING, self.frame.shape[1]-PADDING],
])
self.perspectiveMatrix = cv2.getPerspectiveTransform(markers, boundary)
```

Code snippet 5: Calibration marker detection and transform matrix calculation