CHAPTER-3

Implementation Details

3.1 Generation of ArUco Markers

Before we localize and navigate our robot in 2D space, we first have to uniquely identify it using fiducial markers. The markers used in this project are ArUco markers **[1]** which are simple synthetic square markers composed of a wide black border and an inner binary matrix which determines its ID.

There are several ways to generate ArUco markers. Some even as simple as installing an external module like **ar-markers** and executing **ar_markers_generate.py** from the terminal. One of the most popular ways to generate markers is to use the OpenCV **aruco** library which contains the **cv.aruco.drawMarker()** method.

However each of these methods imposes certain restrictions on the type of markers we can generate at a time. To overcome those hurdles, we have designed a robust web-based ArUco marker generator titled **ArUco-Gen** (shown in Fig 3.1) for creating fiducial markers with fixed or random IDs.

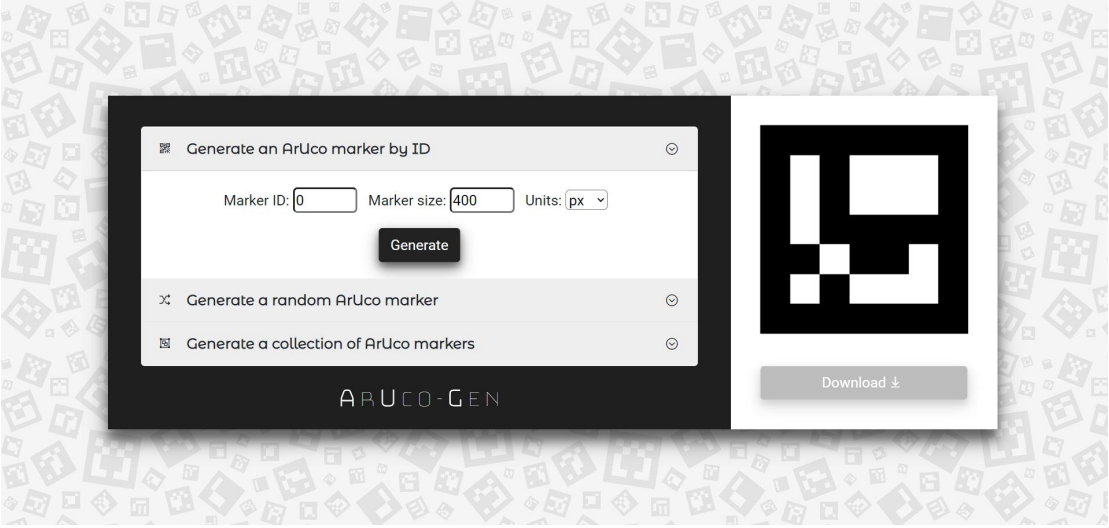

Fig. 3.1. Landing page of the ArUco - Gen web application

Built using Bootstrap and JavaScript (with jQuery on top), ArUco-Gen offers the following features (as shown in Fig. 3.2) -

1. Generate markers with custom and random IDs
2. Generate a collection of markers (upto 500 at a time!)
3. Set marker size in preferred units (px, mm, inches)

Fig. 3.2. ArUco marker generation based on fixed or random IDs [with provisions for selecting marker size and units (px, mm, inches)]

The custom and random markers can be downloaded in the following three formats - PNG, JPEG, SVG. The collection of markers can be obtained in the form of a multi-page PDF document.

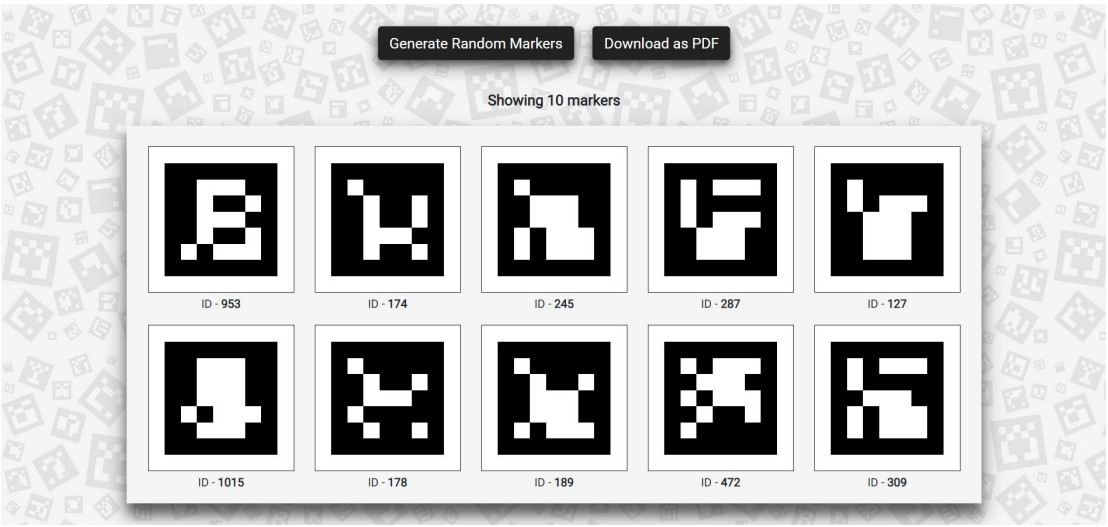

Fig. 3.3. Generating multiple ArUco markers at a time (up to 500 markers allowed)

The binary marker matrix responsible for marker identification has been generated with the help of the **aruco-marker** JavaScript library. The dictionary used for marker generation is the original ArUco dictionary (5x5) [DICT_ARUCO_ORIGINAL]

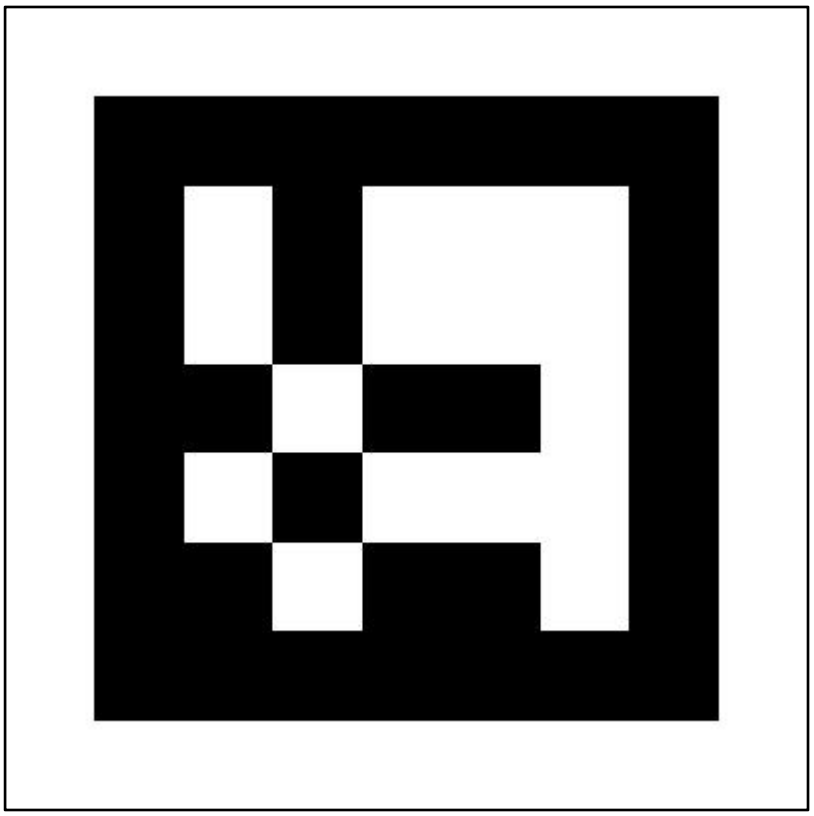

Fig. 3.4. An ArUco marker with an ID of 358 and size of 400px X 400px

Now that we have generated our ArUco markers, the next step in the pipeline is calibrating our cameras and detecting them.

3.2 Camera Calibration

Camera calibration is the process of estimating the intrinsic and extrinsic parameters of a camera. These parameters allow us to map 3D points in the real world to its corresponding 2D projection. Calibration also aids in the process of rectifying distortions in the image.

Most pinhole cameras introduce several distortions to the images they capture. Two major distortions caused by the camera lens are -

1. Radial distortion: as we move further away from an image, straight lines appear to be curved leading to radial distortions (as shown in Fig. 3.5)
2. Tangential distortion: if the camera lens is not aligned perfectly parallel to the imaging plane, some areas may look closer while others distant leading to tangential distortions in the captured image

Fig. 3.5. Barrel and Pincushion distortion in a standard chessboard

In addition to solving these distortions, camera calibration yields the values of the the following parameters necessary for pose estimation and augmented reality applications -

1. Intrinsic parameters - parameters specific to the camera lens like - focal length (f_x,f_y), optical centers (c_x,c_y) and distortion coefficients.
2. Extrinsic parameters - parameters corresponding to rotation and translation vectors that represent the camera position and orientation in 3D space.

In order to calibrate our camera, we need some well-defined patterns (like a chessboard or a ChArUco marker grid) w.r.t which we compute the relationship between real world coordinates and those in 2D space.

The OpenCV method **cv2.findChessboadCorners()** returns the corner points in the image if a pattern is detected [in the order left-to-right, top-to-bottom] over multiple passes (as shown in Fig. 3.6)

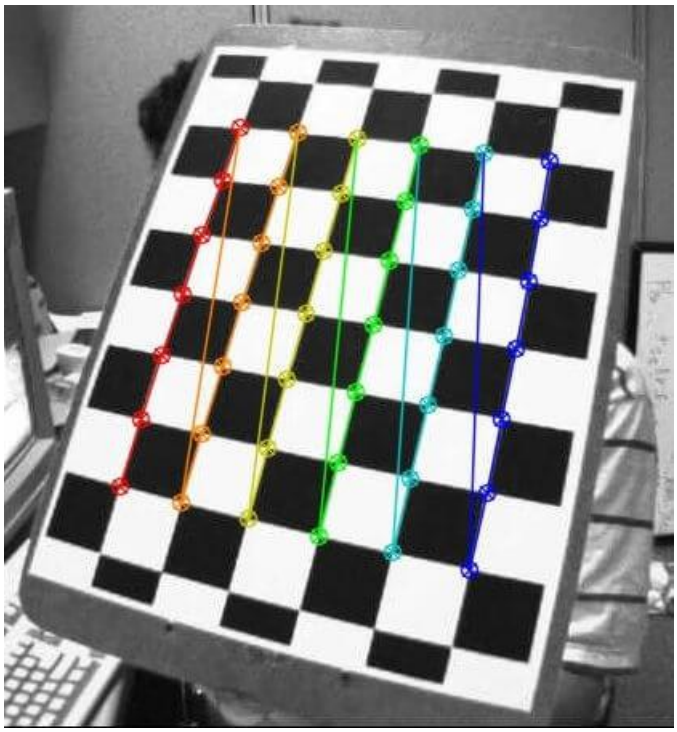

Fig. 3.6. 7x6 test pattern drawn from left-to-right, top-to-bottom

After drawing multiple patterns using **cv2.draw.ChessboardCorners()** to increase our accuracy, we calibrate the camera from the object and image points obtained. The output of this calibration is -

1. cameraMatrix: a 3x3 floating point matrix of intrinsic parameters for, A =

$$\begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$$

2. distortionCoefficients: a vector of distortion coefficients of 4, 5, 8 or 12 elements
3. rvec: a Rodrigues rotation vector, or axis-angle representation
4. tvec: translation vector estimated for each pattern view

We can now undistort the image using cv2.undistort() or remapping and solve re-projection errors if any (as shown in Fig. 3.7)

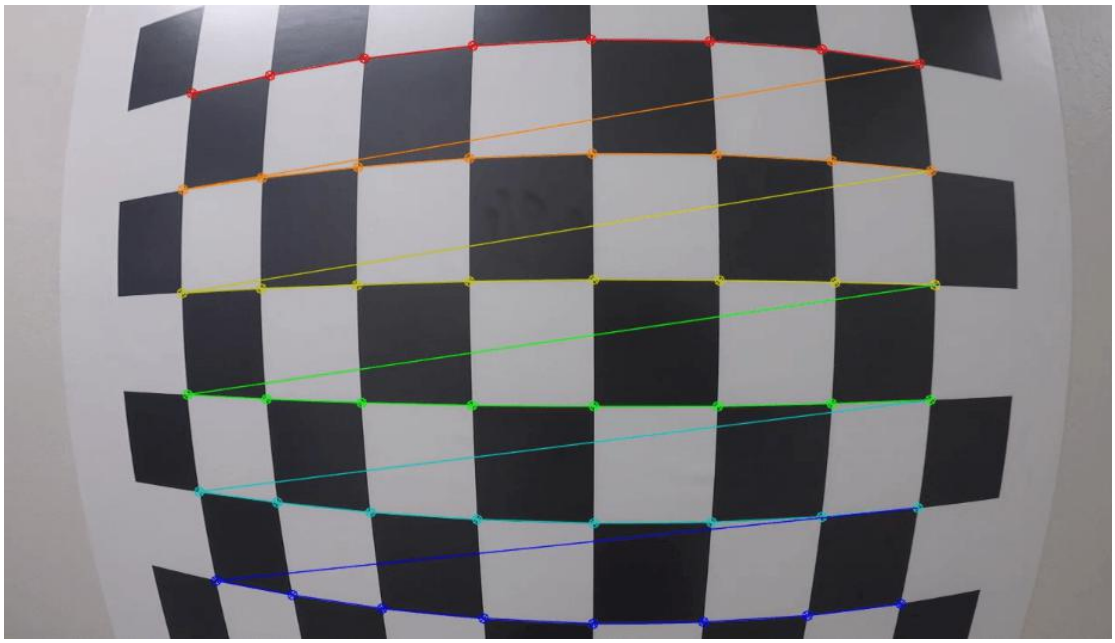

Fig. 3.7. Using either cv2.undistort() or remapping to undistort the image

The camera calibration methods in the OpenCV library has been implemented based on a paper **[7]** by Zhengyou Zhang.

3.3 Detection of ArUco Markers

Given an image or a video feed containing ArUco markers, the detection process has to return a list that contains each marker identified by their IDs (as shown in Fig. 3.8). Each detected marker includes:

1. The coordinates of the four corners of the marker in the original order (top-left, top-right, bottom-right, bottom-left in a clockwise manner)
2. The id of the marker depending upon the dictionary used

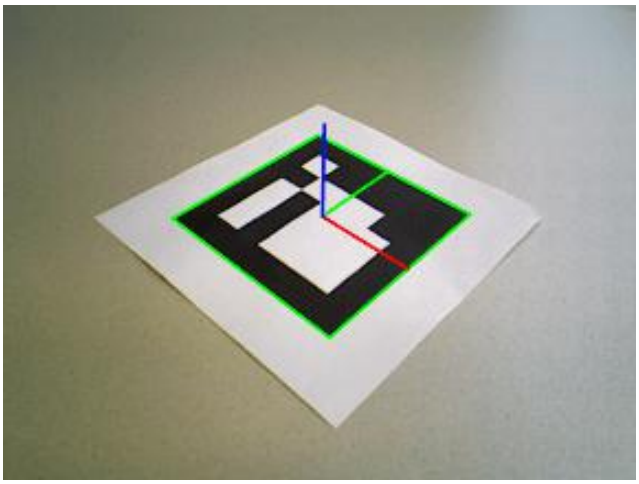

Fig. 3.8. An ArUco marker with a green bounding box and three axes drawn at the center (shown using three different colors - x: red, y: green, z: blue)

There are several camera applications that can be used to detect markers, the most commonly used being webcams or mobile applications that offer similar features like IP webcam (broadcasting over a network).

Nowadays, smart phones are perfectly capable of processing video feed which is exactly what we employed. We modified an open source project called **Aruco Android Server** to detect corners and calculate the center coordinates, the heading (in radians) and the size of the marker before sending the response to the client. This new application **ArUco Scanner** improves on the previous methods and reduces latency caused due to compression, transmission and decompression.

Fig. 3.9. ArUco Scanner Android application (server running at 192.168.2.2:5000)

When the ArUco Scanner server receives a request with the character “g” encoded with it, it sends a JSON response which looks like this:


```
{
   "aruco": [
      {
         "ID": 2,
         "center": {
            "x": 266,
            "y": 259
         },
         "heading": -3.1150837133093243,
         "markerCorners": [
            {
               "x": 129,
               "y": 125
            },
            {
               "x": 396,
               "y": 129
            },
            {
               "x": 402,
               "y": 381
            },
            {
               "x": 137,
               "y": 401
            }
         ],
         "size": 267
      }
   ]
}
```

Thus, the ArUco Scanner app can accurate detect markers in the field of view of the smart phone camera lens and return the computed data in the form of a JSON reponse to the client so that the overhead of calculations on the client-side is reduced (as shown in Fig. 3.10).

Fig. 3.10. Five random ArUco markers detected using the ArUco Scanner app. The markers are bounded with an orange box with the heading indicated with a purple line. The IDs are marked at the center. The server runs at 192.168.2.2:5000 and sends a JSON response to the client containing the marker ID, center and corner coordinates, heading (in radians) and size.

3.4 Localization of ArUco Markers

Processing the video feed for obtaining marker orientation (pose estimation)

Given a marker image, we can utilize the intrinsic and extrinsic camera properties as well as the marker corner coordinates to calculate its pose, or how the object is situated in space, (like how it is rotated, how it is displaced etc).

When we the camera intrinsics (camera matrix and distortion coefficients obtained by calibration) is known, and the size of the marker is specified, the OpenCV library can be used to find the relative position of the markers and the camera. It is given by the *rvec* and t*vec* vectors. They represent the transform from the marker to the camera.

The function **estimatePoseSingleMarkers()** is used to calculate the rotation vector rvec = [a b c].

Using rvec and tvec obtained from camera calibration, we transform a 3D point (in this case our marker mounted robot) expressed in one coordinate system to another one [in our case, from the tag frame to the camera frame].

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{cam} = \begin{bmatrix} {}^{cam}\mathrm{R}_{tag} & {}^{cam}\mathrm{t}_{tag} \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{tag}$$

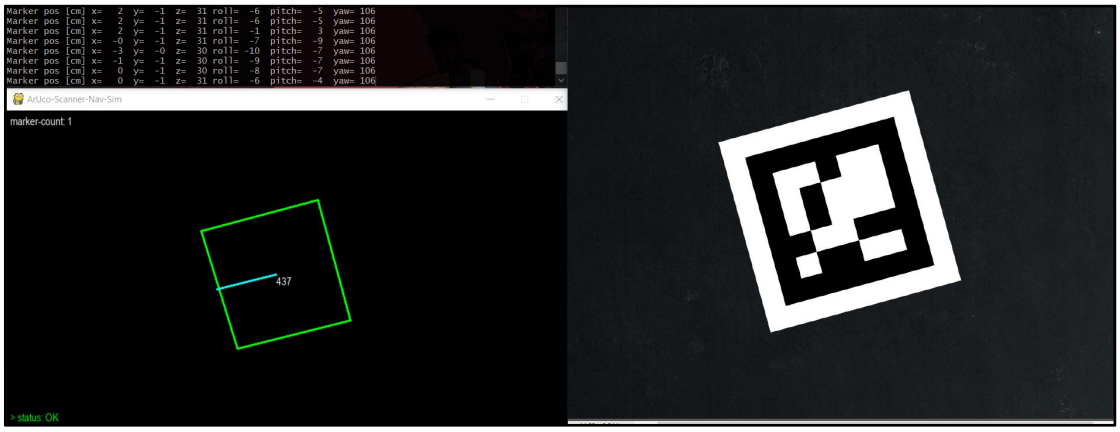

Fig. 3.11. Pose estimation of an ArUco marker

The following figures Fig 3.12 and Fig. 3.13 show the marker attitude on three faces of an ArUco cube by computing the position of its points **relative** to the camera. This is why marker rotation, skewing or distortions do not affect its detection in any way.

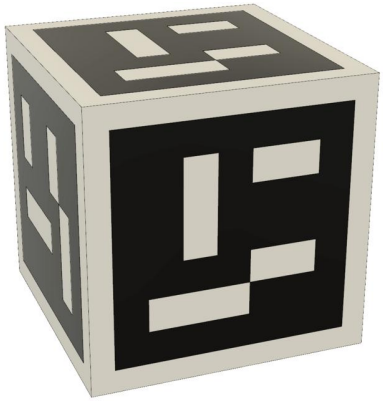

Fig. 3.12. An ArUco marker cube for a sample orientation test

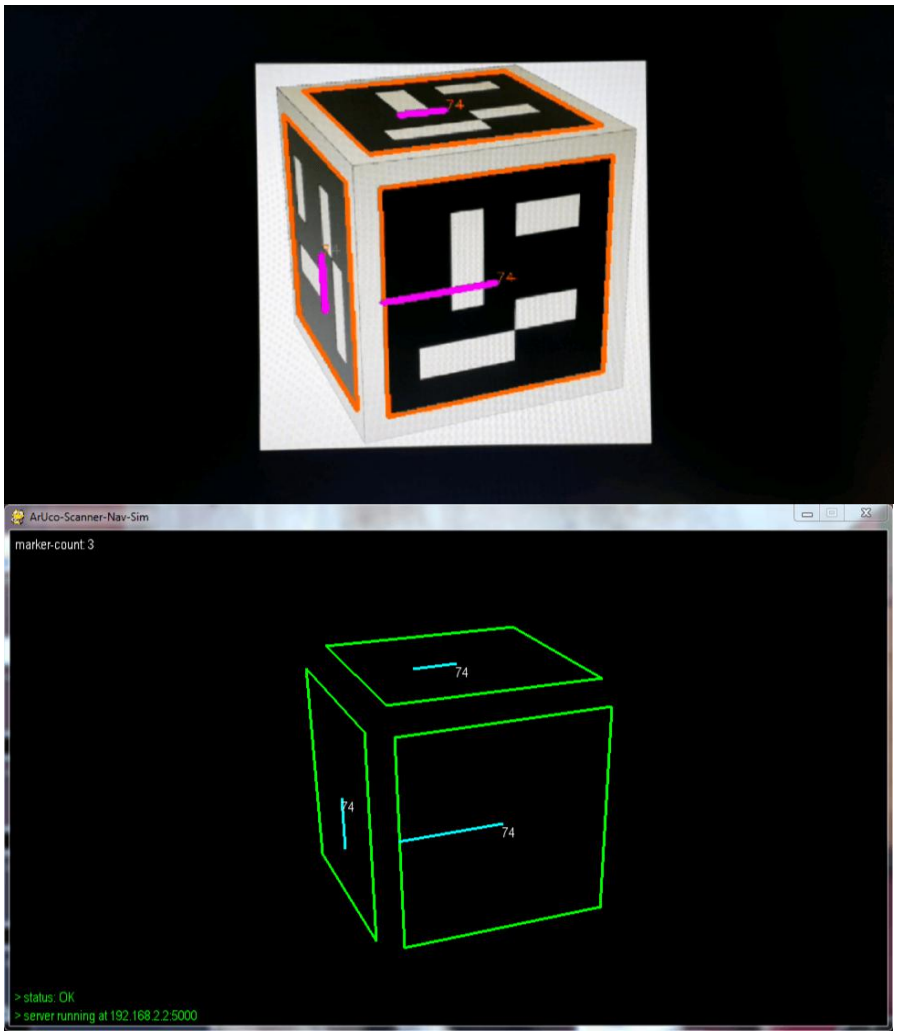

Fig. 3.13. Scanned output (top) and localization with distinct orientation (bottom)

The ArUco scanner application was used to detect the markers on a screen and send the JSON response containing marker data. A pygame window was used to simulate the localization process based on the JSON data (as shown in Fig. 3.14).

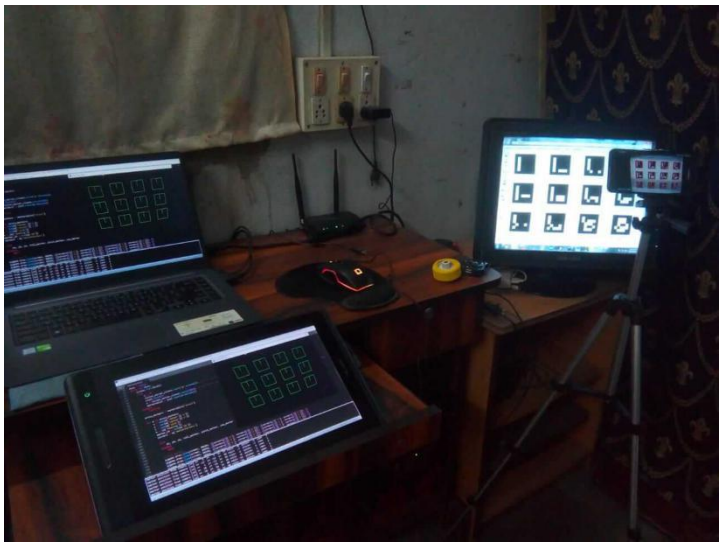

Fig. 3.14. ArUco marker localization setup (with ArUco Scanner running on an Android smart phone and multiple random ArUco markers displayed on a screen)

The algorithm of the localization program (**aruco_nav_sim.py**) is as follows:

Step 1: Start

Step 2: Import required modules and create a socket instance

Step 3: Initialize pygame with a window size of 1280x720

Step 4: Establish connection to vision server at 192.168.2.2:5000

Step 5: If window is not closed, goto Step 6, else goto Step 15

Step 6: Send a request to the server with the character "g" encoded

Step 7: Load the JSON response for all markers in a new list

Step 8: For every marker in the list, do Step 9 to Step 14

Step 9: Extract the marker ID, size, heading, corner and center coordinates

Step 10: Call getMarkerOrientation() method to get marker attitude

Step 11: Display the values of x_m, y_m, z_m coordinates and roll, pitch and yaw

Step 12: Draw bounding box on each marker using the corner coordinates

Step 13: Draw the heading using the center coordinates

Step 14: Update the screen

Step 15: Exit

The following figures illustrate the localization process and reflects the state of the markers at each step of the work-flow (as shown in Fig. 3.15, Fig. 3.16 and Fig. 3.17).

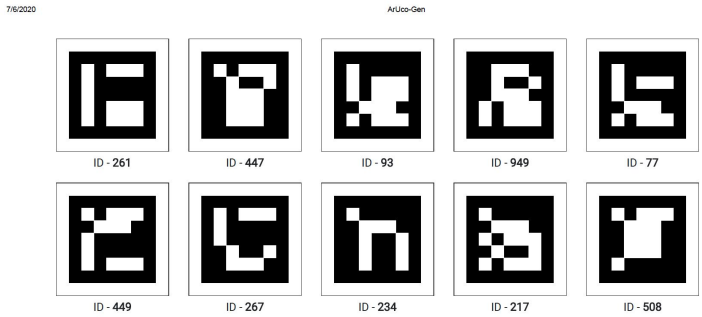

Fig. 3.15. Ten random ArUco markers generated using the ArUco-Gen web app

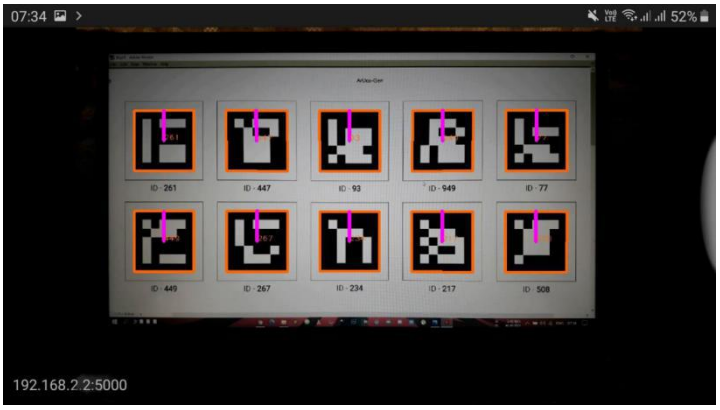

Fig. 3.16. Bounding boxes and heading lines generated for the ten markers using the ArUco Scanner Android application. The JSON response sent by the server to the client contains marker id, size, coordinates and heading (in radians)

Fig. 3.17. The pygame window plots the detected markers by surrounding each with a bounding box (in green), a heading line (in cyan) and a marker ID (in white). The marker attitude in terms of the x_m,y_m,z_m coordinates and the roll, pitch and yaw of the ArUco markers are logged in the terminal.

The following Fig. 3.18 shows the entire work-flow up to localization in one frame with sliced windows each showing one step of the whole process.

The right most window shows a stack of 20 random ArUco markers (generated using the ArUco-Gen web-app) displayed in PDF format.

The top left window shows the translational vector values (tvec = [x_m, y_m, z_m]) and rotational vector values (in terms of Euler 321 angles - roll , pitch and yaw) being logged on a terminal.

The bottom left window shows the actual localization of all 20 markers with their positions reflecting those of the scanned feed on the smart phone through the ArUco Scanner application. The pygame window also shows the marker-count (number of markers on the screen), connection status and the server IP it is connected to during the localization.

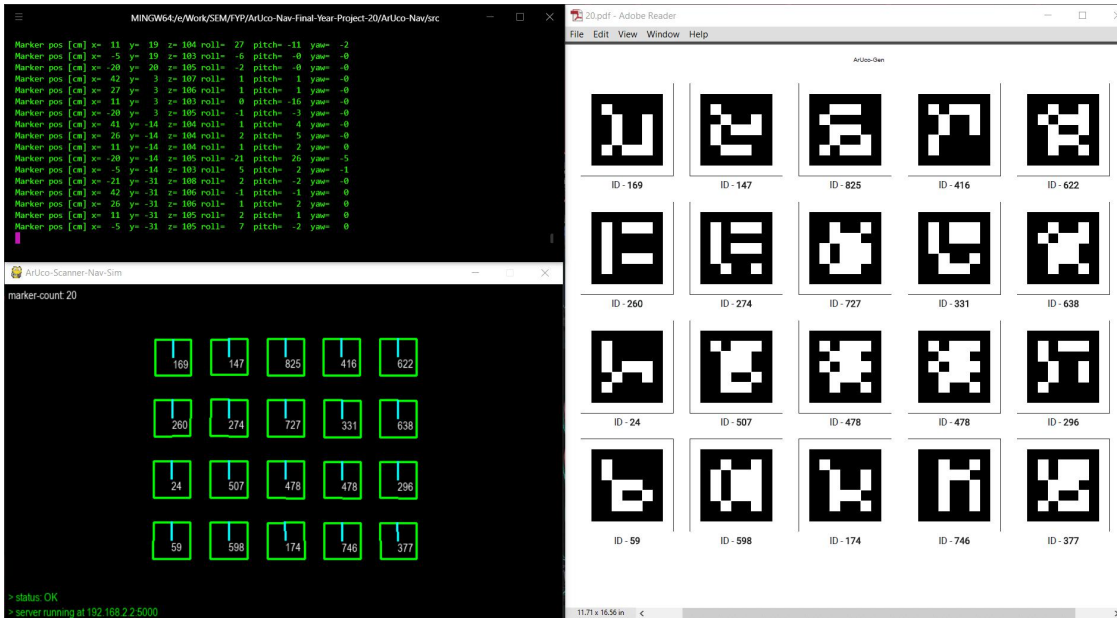

Fig. 3.18. A consolidated representation of the localization process - (from the right clockwise), a set of 20 ArUco markers, the marker orientation (attitude) data being printed on the terminal, the pygame window showing the positions of all 20 markers

3.6 Visual Navigation (Simulation) of ArUco Markers

We have partially simulated a waypoint navigation system among swarm robots by representing each robot as an ArUco tag itself. The marker attitude in terms of the x_m, y_m, z_m coordinates and roll, pitch and yaw are used to orient the robot (in our case, the marker) towards the required heading (in our case, towards a target). The locomotion is simulated in a pygame window (as shown in Fig. 3.19 and Fig. 3.20).

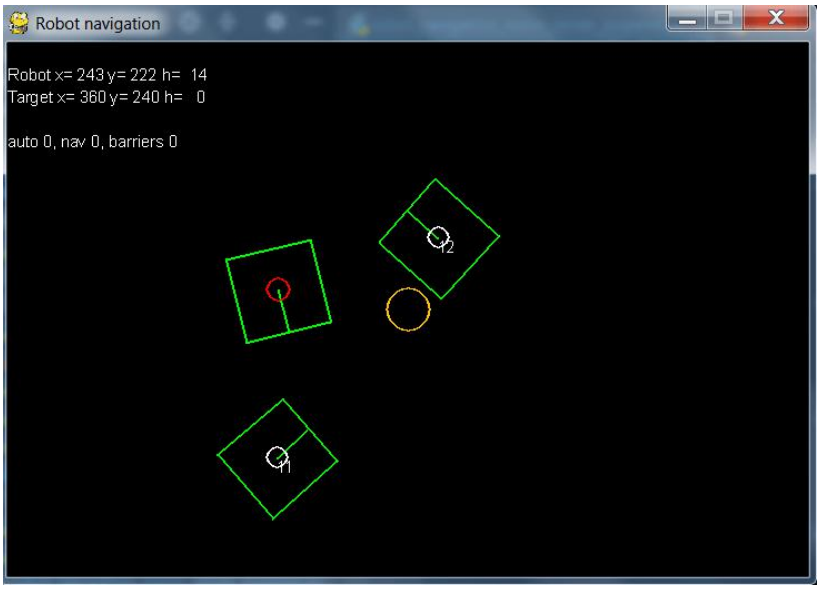

Fig. 3.19. A test simulation showing a robot (marked with a red circle at the center), two barrier objects (marker IDs 11 and 12) and a target (yellow circle)

Fig. 3.20. A consolidated representation of the navigation process showing the ArUco markers (right), marker attitude (top-left), and simulation (bottom-left)

A detailed report on the above simulation has been made in the outputs section below.

System Validation

The system was validated to assure the compliance of the following elements - ArUco-Gen (marker generator), ArUco Scanner (marker detector) and the ArUco localization and navigation simulator (as shown in Fig. 3.21)

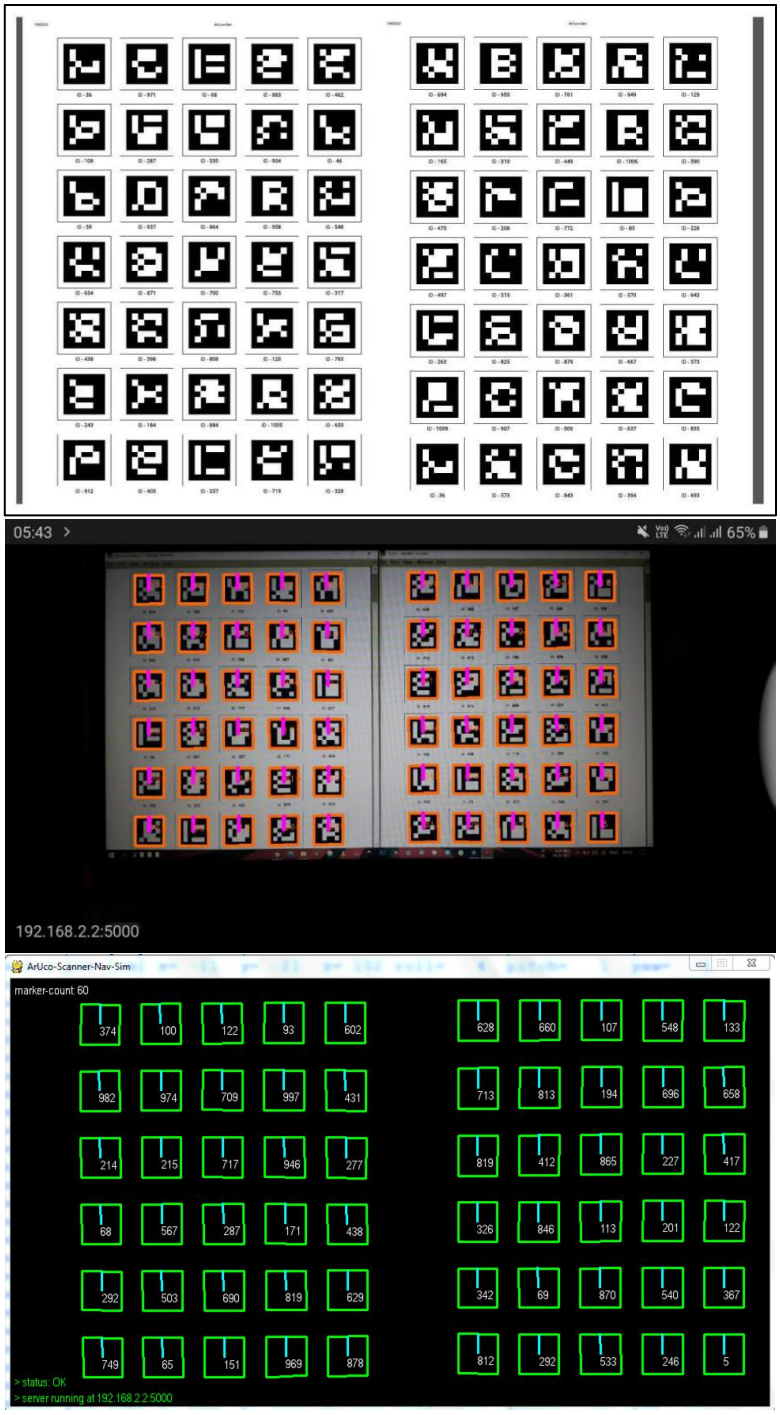

Fig. 3.21. System validation [from top, ArUco-Gen (generator), ArUco Scanner (detector) and ArUco Scanner Nav Sim (localization and navigation simulator)]

Observed Output

For simulating the navigation and logging the results, a simple 15 seconds scene was arranged with 3 swarm robots (indicated by their green bounding boxes and their ArUco marker IDs generated using **ArUco-Gen**) and 4 barrier points (indicated by red bounding boxes).

A total of 60 frames were recorded and scanned at the rate of 8fps using the overhead smart phone camera running **ArUco Scanner** server at 192.168.2.2:5000. The response obtained from the server was processed to log the marker (and hence, robot) orientation in 2D space (using the tvec and rvec components - x_m, y_m, z_m, roll, pitch and yaw). The final simulation was shown in the pygame window **ArUco Scanner Nav Sim** (shown in Fig. 3.22).

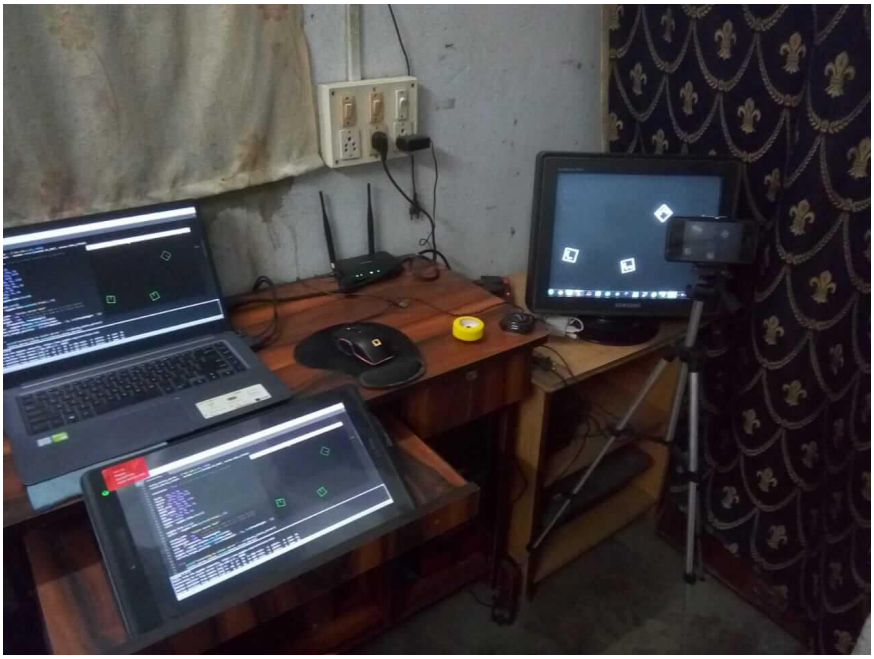

Fig. 3.22. A simple localization and navigation setup using ArUco Scanner on a smart phone mounted on a tripod and the simulation logged on the laptop screen

Out of the 60 frames captured, here we show 5 frames which indicate the motion of the 3 ArUco markers (and by extension, the 3 swarm robots) at intervals of 10 frames. [The three images in each output are: (from top) ArUco markers (swarm robots representation), scanned output (from the app) and the final simulation. The time-stamp and frame number of each output is given below each figure.]

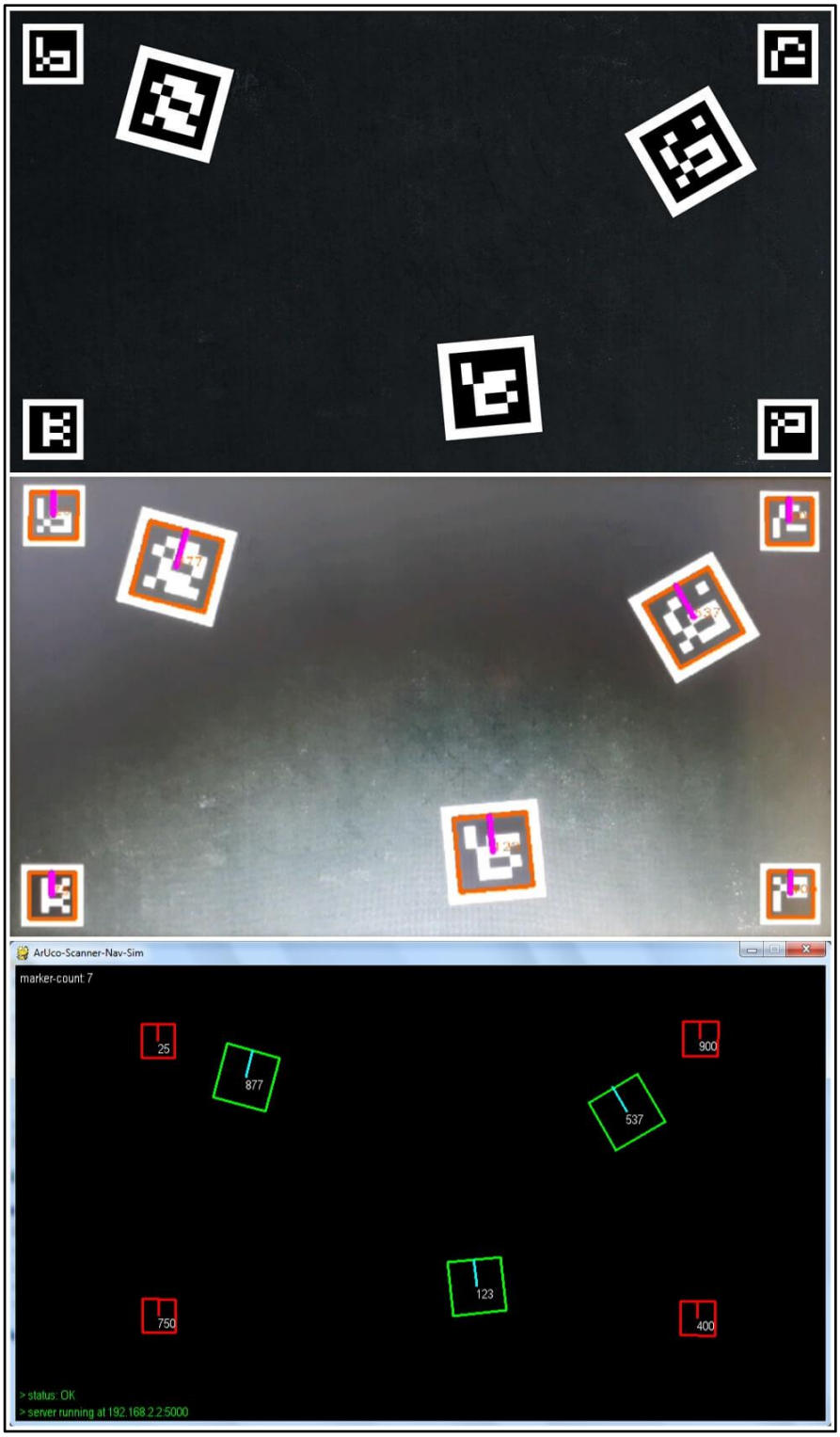

Fig. 3.23. output - time-stamp: 00:01:25; frame no.: 5

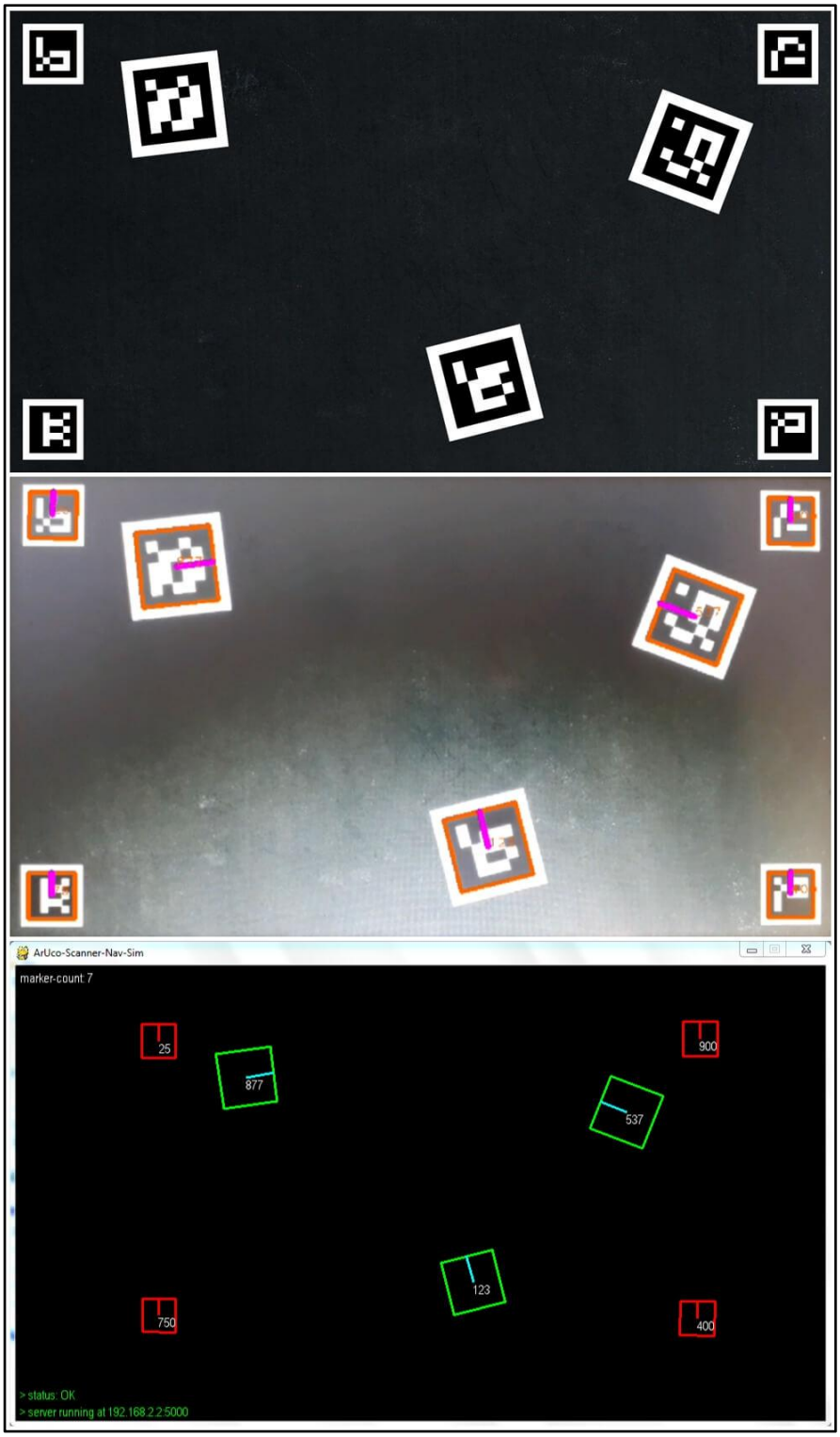

Fig. 3.24. output - time-stamp: 00:03:75; frame no.: 15

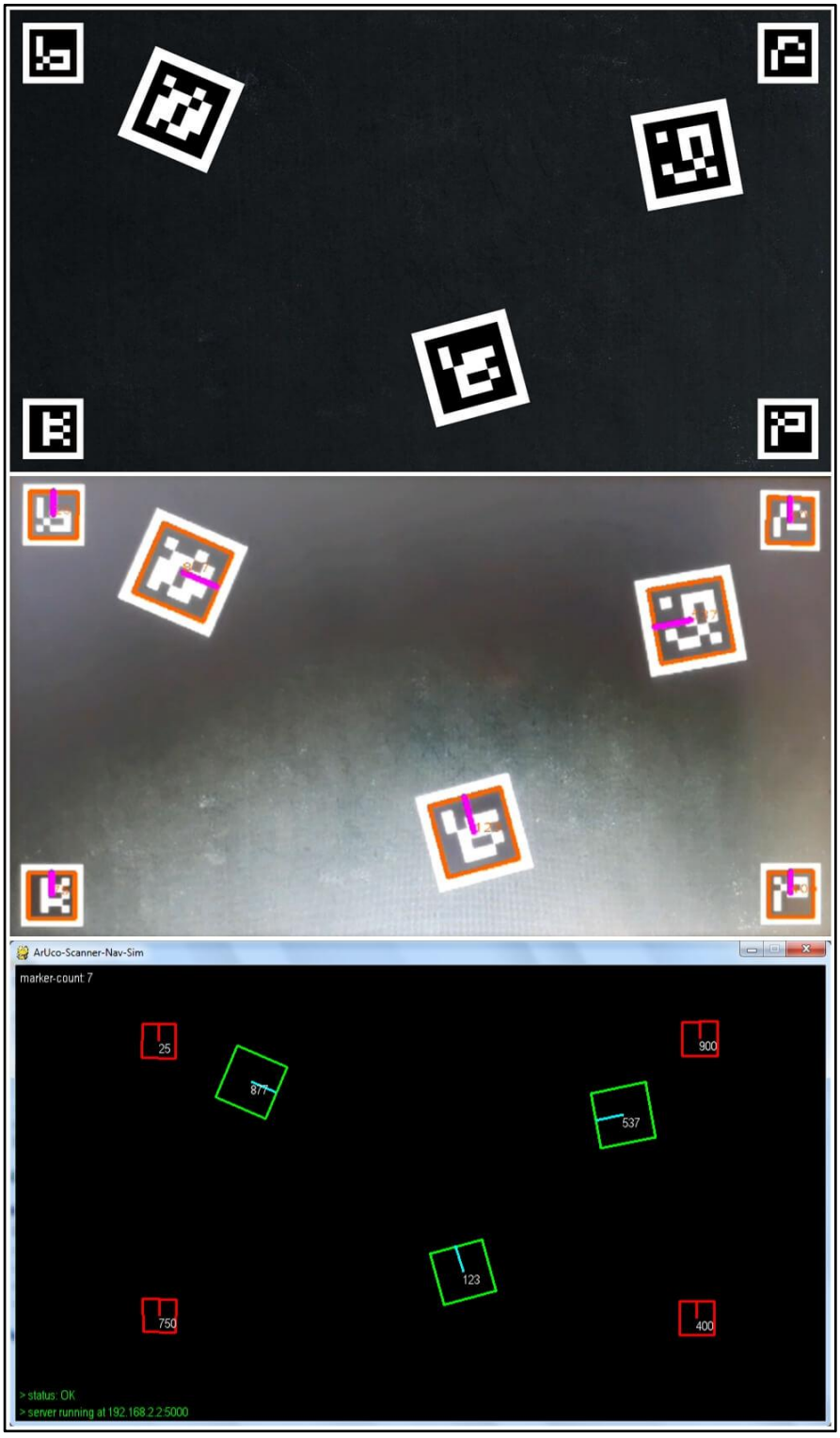

Fig. 3.25. output - time-stamp: 00:06:25; frame no.: 25

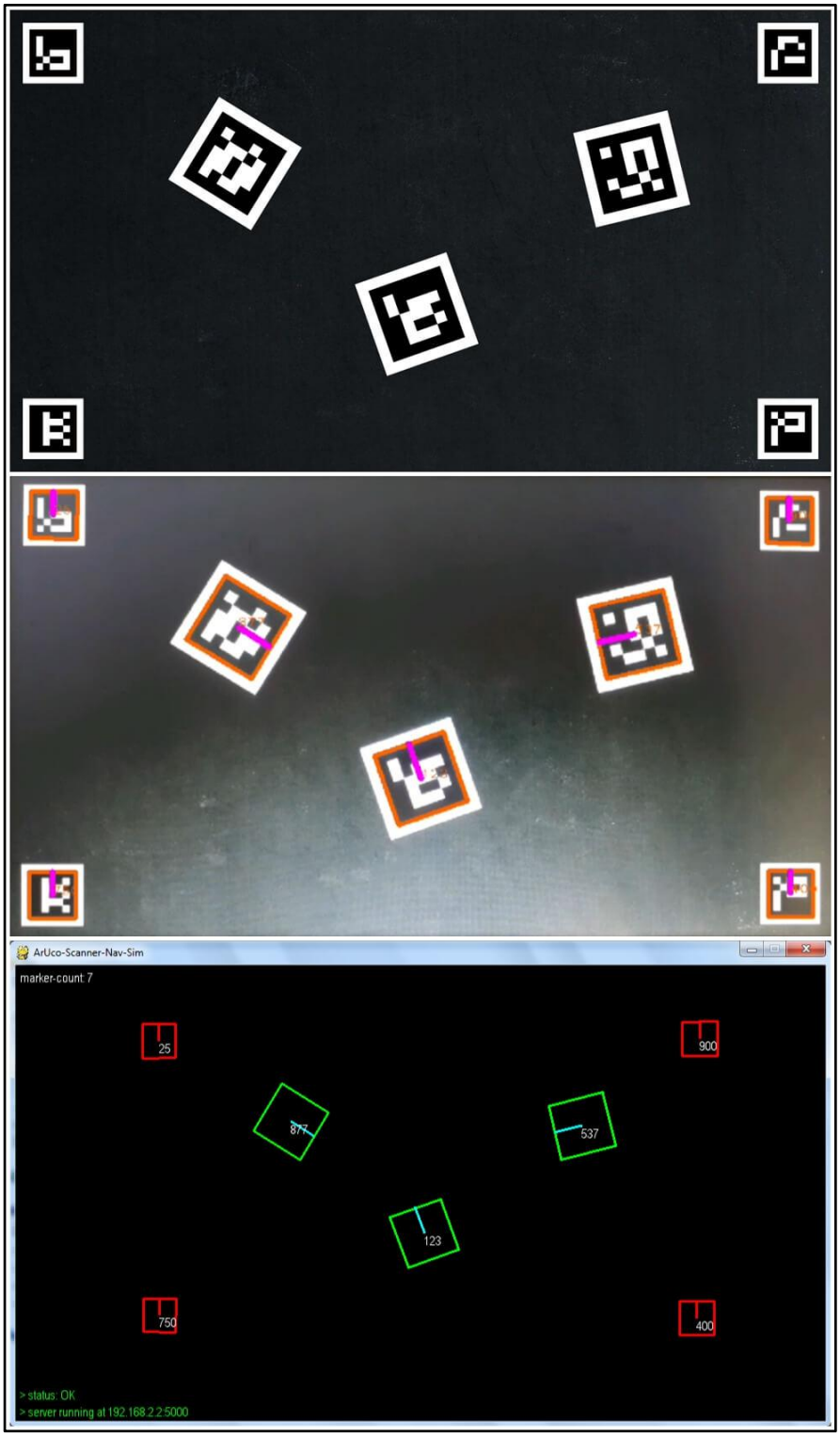

Fig. 3.26. output - time-stamp: 00:11:25; frame no.: 45

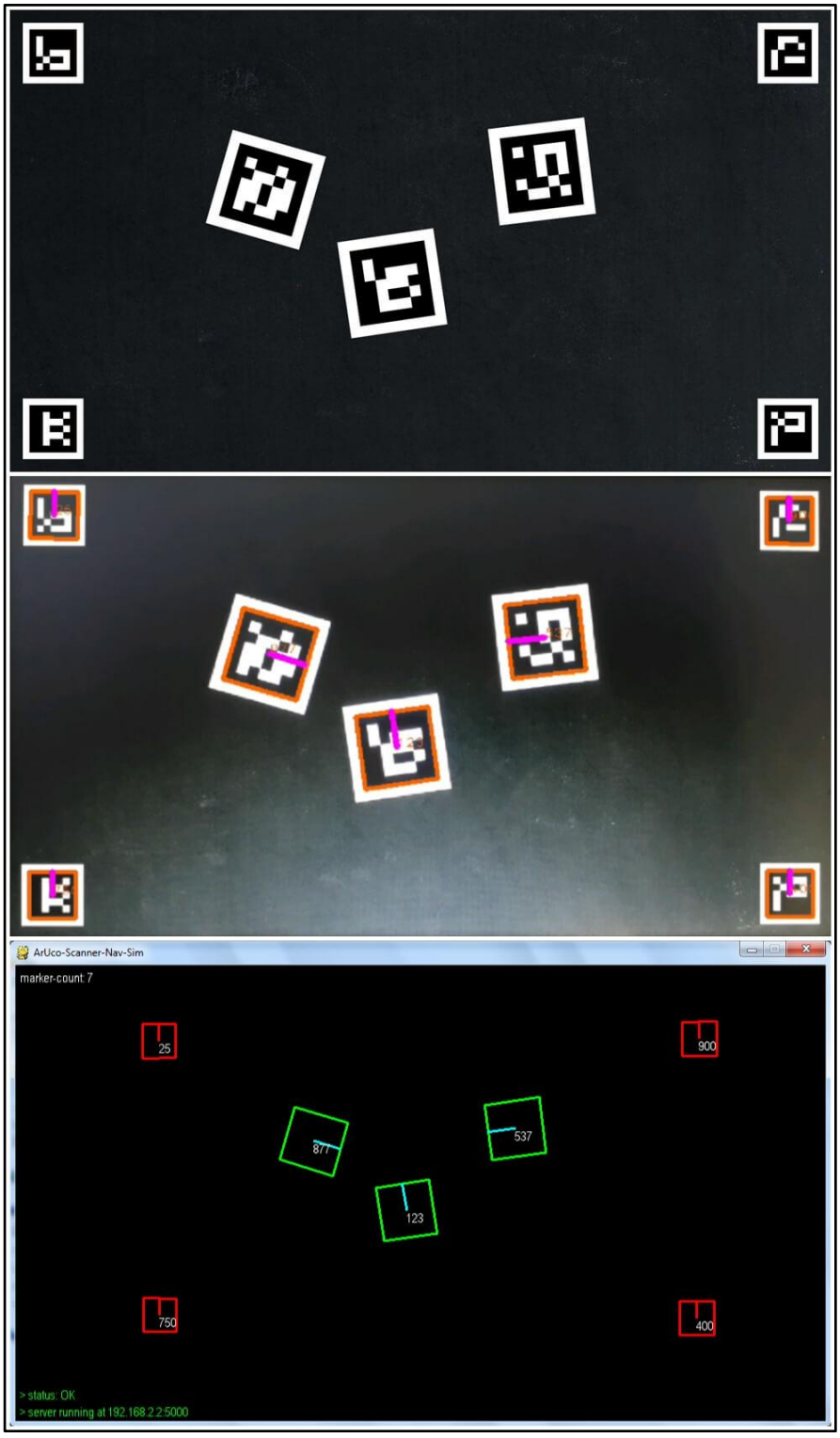

Fig. 3.27. output - time-stamp: 00:13:75; frame no.: 55

Result Discussion and Analysis

1. The final simulation prototype was successful in detecting, localizing and tracking ArUco markers, and by extension, a scalable robot swarm.
2. The camera calibration and subsequent pose estimation performed are in accordance with the results shown in the outputs section above.
3. The rotation, translation or skewing of any marker did not affect its detection, thus proving that the marker orientation parameters were near accurate.
4. There was a slight latency in the simulation compared to the detection on the server side and this might have been due to the network they were connected on.
5. The detection depends heavily on the lighting conditions as markers visible to the naked eye were not detected by ArUco Scanner in a dimly lit room.
6. The detection process is also affected by the size of the markers and the relative distance between the marker and the camera.
7. Way point navigation was partially realized through a simple 15 seconds 60 frames simulation scene yielding mostly positive results.
8. The speed of marker locomotion also affected their detection as bounding boxes were missing for few frames.
9. The processed video maps the exact resolution of the scanner onto the frame window of the pygame application; thus higher resolutions would be preferred.
10. The application was clearly taxing on the smart phone as it reached approximately 8-10 frames per second.

FUTURE SCOPE

The simulation of the waypoint navigation was successfully implemented, but we aim to interface an actual robot capable of communicating with the localization and navigation components using suitable message broker protocols (MQTT/TCP).

The intended behaviour of the ArUco-Nav system was for each swarm robot to autonomously manoeuvre between predetermined or dynamic way points (as shown in Fig.3.28) through the use of visual communication with an overhead camera and local communication with other robots.

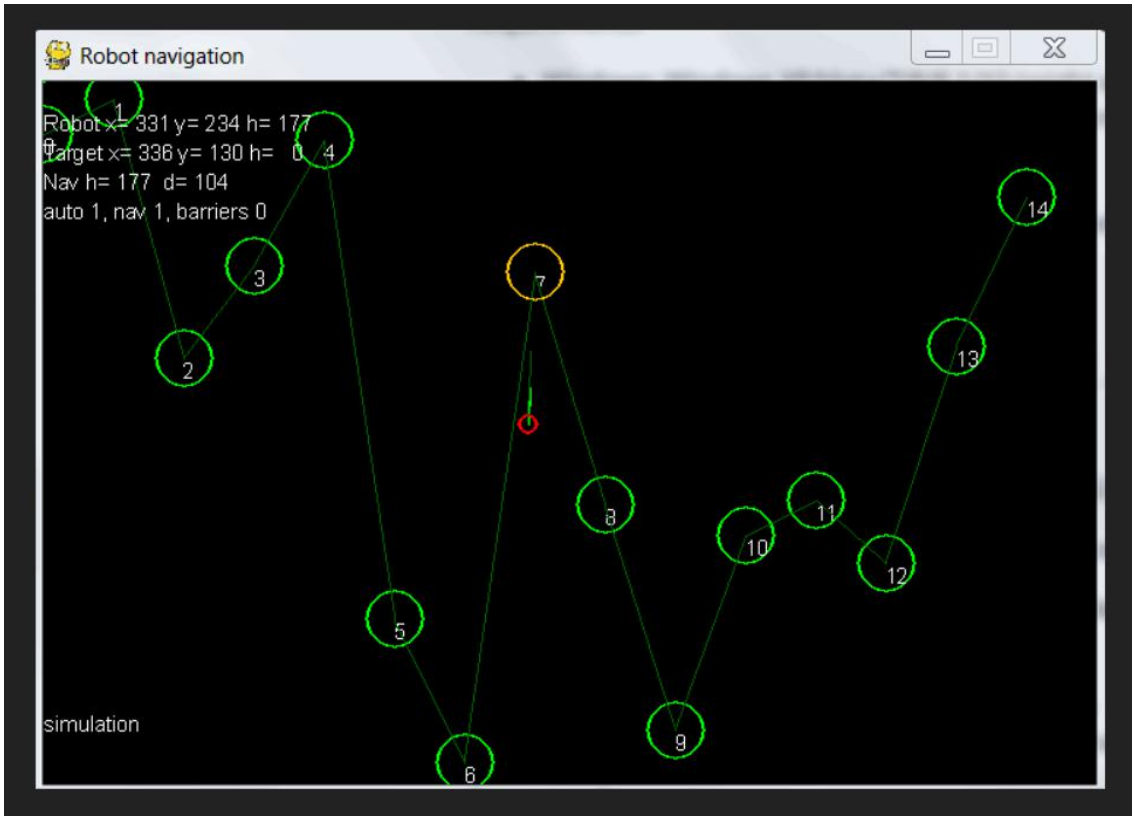

Fig. 3.28. Intended simulation of way-point navigation of a swarm of robots with barriers and dynamic way-points.

After interfacing a single robot, we eventually want to extend it to a swarm of robots with common way points but autonomous navigation (vision-guided) in a controlled environment. Each swarm robot will be able to wireless communicate with the ArUco scanner server and each other through the use of an IoT development board like NodeMCU or BoltIoT.

Fig. 3.29. A simple 2WD robot build using Arduino Nano, L298N and BO-motors

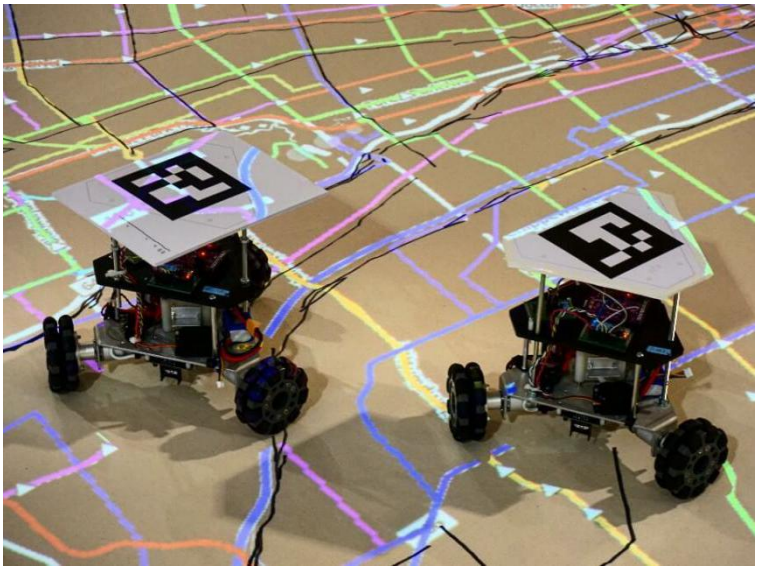

Fig. 3.30. A swarm robot configuration with ArUco markers mounted on each robot

CONCLUSION

Localization and navigation of mobile robots in 2D space is indeed a challenging task but with more emerging technologies like ArUco and other fiducial markers, vision based detection and pose estimation is now achievable at a large scale.

Through our project ArUco-Nav, we have successfully achieved an appropriate simulation of a robot (or a robot cluster) detection using ArUco tags. Localization of their coordinates in 2D space was achieved through the use of overhead cameras. Image processing capabilities through the OpenCV python library aided in camera calibration and accurate orientation (attitude) computation. This in turn improved the overall navigation mock-up and yielded precise detection results.

With a few test cases, it had become abundantly clear that the accuracy in localization is affected by the size and amount of visible markers, as well as the distance between the detector used and the marker itself.

Proper camera calibration can tune these inaccuracies to a good extent and yield near precise marker orientation parameters. Navigation as a supplementary functionality to localization was similarly affected due to marker readability factors. However, all cases proved that ArUco markers can be used a visual markers for the tracking and guidance of a scalable robot swarm in 2D space.

REFERENCES

[1]. Avola, Danilo & Cinque, Luigi & Foresti, G.L. & Mercuri, Cristina & Pannone, Daniele. (2016). A Practical Framework for the Development of Augmented Reality Applications by using ArUco Markers. 645-654. 10.5220/0005755806450654.

[2]. Romero-Ramirez, Francisco & Muñoz-Salinas, Rafael & Medina-Carnicer, Rafael. (2018). Speeded Up Detection of Squared Fiducial Markers. Image and Vision Computing. 76. 10.1016/j.imavis.2018.05.004.

[3]. Garrido-Jurado, Sergio & Muñoz-Salinas, Rafael & Madrid-Cuevas, Francisco & Medina-Carnicer, Rafael. (2015). Generation of fiducial marker dictionaries using Mixed Integer Linear Programming. Pattern Recognition. 51. 10.1016/j.patcog.2015.09.023.

[4]. Romero-Ramirez, Francisco & Muñoz-Salinas, Rafael & Medina-Carnicer, Rafael. (2019). Fractal Markers: a new approach for long-range marker pose estimation under occlusion. 10.13140/RG.2.2.31185.17767.

[5]. Babinec, Andrej & Jurišica, Ladislav & Hubinský, Peter & Duchoň, František. (2014). Visual Localization of Mobile Robot Using Artificial Markers. Procedia Engineering. 96. 10.1016/j.proeng.2014.12.091.

[6]. Alves, Paulo & Costelha, Hugo & Neves, C.. (2013). Localization and navigation of a mobile robot in an office-like environment. 1-6. 10.1109/Robotica.2013.6623536.

[7]. Zhengyou Zhang (2000). A Flexible New Technique for Camera Calibration. IEEE Trans. Pattern Anal. Mach. Intell. 22(11): 1330-1334 (2000)