Localization and Visual Navigation of a Scalable Robot Swarm using ArUco Markers

by

Name	Roll No.	Registration No:
Swarnendu Sardar	**11700116026**	**161170110087** of 2016-2017
Sounak Mondal	**11700116035**	**161170110078** of 2016-2017
Soumik Dhar	**11700116039**	**161170110074** of 2016-2017
Arindam Samanta	**11700116098**	**161170110015** of 2016-2017

A comprehensive project report has been submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

COMPUTER SCIENCE & ENGINEERING

Under the supervision of

Dr. Pramit Ghosh
Assistant Professor
Dept. of Computer Science & Engineering
RCC Institute of Information Technology

Department of Computer Science & Engineering
RCC INSTITUTE OF INFORMATION TECHNOLOGY
Affiliated to Maulana Abul Kalam Azad University of Technology, WestBengal
CANAL SOUTH ROAD, BELIAGHATA, KOLKATA – 700015

July, 2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
RCC INSTITUTE OF INFORMATION TECHNOLOGY

TO WHOM IT MAY CONCERN

I hereby recommend that the project entitled "**Localization and Visual Navigation of a Scalable Robot Swarm using ArUco Markers**" prepared under my supervision by Swarnendu Sardar (11700116026), Sounak Mondal (11700116035), Soumik Dhar (11700116039) and Arindam Samanta (11700116098) may be accepted in partial fulfillment of the requirements for B.Tech degree in **Computer Science & Engineering** from **Maulana Abul Kalam Azad University of Technology, West Bengal.**

P Ghosh
Digitally signed by P Ghosh
DN: cn=P Ghosh, c=IN, o=RCCIIT, ou=CSE
Date: 2020.07.07 17:09:16 +05'30'

.....
Project Supervisor
Department of Computer Science and Engineering
RCC Institute of Information Technology

Countersigned:

.....
Head
Department of Computer Sc. & Engg,
RCC Institute of Information Technology
Kolkata – 700015.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
RCC INSTITUTE OF INFORMATION TECHNOLOGY

CERTIFICATE OF APPROVAL

The foregoing Project is hereby accepted as a credible study of an engineering subject carried out and presented in a manner satisfactory to warrant its acceptance as a prerequisite to the degree for which it has been submitted. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve the project only for the purpose for which it is submitted.

FINAL EXAMINATION FOR
EVALUATION OF PROJECT

1. …….

2. …….

3. ……..……

(Signature of Examiners)

ACKNOWLEDGEMENT

We would like to take this opportunity to express our profound gratitude and deep regards to our mentor – Dr. Pramit Ghosh who gave us the golden opportunity to work on this project and also for all his guidance, mentoring and constant encouragement throughout the course of this project.

We would also like to extend our gratitude to all the people for their direct or indirect involvement towards the completion of this project. Also, not to forget the coordinated efforts put forward by our work and hereby we thank each other for the successful completion of the documentation.

Arindam Samanta (CSE2016/081)

Sounak Mondal (CSE2016/083)

Soumik Dhar (CSE2016/084)

Swarnendu Sardar (CSE2016/087)

Abstract: This project deals with one of the most interesting problem in the field of robotics - localization and navigation of a mobile robot. The ability of a robot to find its location in the form of x and y coordinates in the 3D space or effectively in the 2D space and then finding its way to reach the destination is a very common problem and is also a very computation intensive and complex task. It has to take into account not just its own position relative to a reference point in the 2D space but also calculate the distance between the destination and its current position, as well as do the same for any obstacles that might come in its way. Not to mention the great applications that they provide, autonomous robot navigation can be used from regular household works to heavy duty industrial work.

Keywords: Localization, Navigation, Indoor Navigation Systems, Fiducial Markers, ArUco, Swarm Robotics

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ArUco:	Augmented Reality University of Cordoba
Bot:	Robot
Nav:	Navigation
OpenCV:	Open Source Computer Vision library
VBL:	Vision Based Localization
MQTT:	Message Queuing Telemetry Transport
GPS:	Global Positioning System
TCP:	Transport Control Protocol
PID:	Proportional, Integral, Derivative
Contrib:	Contributor
2WD:	2-wheel drive
IoT:	Internet of Things
BO-motors:	Battery Operation motors
Li-ion:	Lithium-ion
Json:	JavaScript Object Notation
Lib:	Library
Rad:	Radian
AR-marker:	ArUco Marker
px:	Pixel
rvec:	Rotation vector
tvec:	Translation vector

CHAPTER-1

Introduction

Navigation is one of the most demanding capabilities required of a mobile robot. The four building blocks of navigation are:

Perception- the robot must interpret its sensors to extract meaningful data;

L***ocalization***- the robot must determine its position in the environment;

Cognition- the robot must decide how to act to achieve its goals; and

Motion-control- the robot must modulate its motor outputs to achieve the desired trajectory.

Of these four components, localization has been the focus of several research advances done in the past decade. Localization can be very easily and cost effectively achieved by using GPS, but given the poor accuracy of GPS in small spaces and that to in indoor conditions we had to come up with something different and something which can be easy to setup and cost effective as well.

Now a days, each and every smart phone has that capacity and capability to carry out extremely complex and compute intensive tasks. So we're using the smart phone camera sensor as the primary driver for - detecting the positions of the bot in 3D space, finding out the 2D coordinates of the bot and then transmitting those coordinates to the server or our PC where the distance between the bot and the destination is computed and transmitted to the bot.

Now the second and most important problem that needs to be addressed is actually identifying the bot in 3D space. This can be attained with the help of image processing, to differentiate and identify the bot in the environment and this process is made simpler by using ArUco markers and Python's OpenCV library for image processing.

After the initial setup is completed, the chassis of the bot is readied, the electronics are mounted and the ArUco markers are pasted on top of the bot so that the camera can distinctly identify the marker and instructions can be relayed to a particular bot using a message broker service like MQTT.

Review of Literature

Navigation systems have countless applications in real life. It is extensively used for finding an entity's location and for navigating from one position to the next. GPS is extremely useful for outdoor positioning and navigation application where accuracy does not matter that much, but for indoor navigation we cannot rely solely on GPS to give us an accurate location.

1.1 Fiducial marker-based detection (over other detection methods)

There are a lot of alternate solutions to this problem through which we can achieve our objective of indoor navigation system. Some of these solutions include - LiDAR sensors, IR sensor or vision-based positioning which utilizes one or more camera sensors. Although these are viable solutions, but they are too expensive to work with. So instead of using any extra hardware we can use our smart phone camera which would act as the eyes for our bot and keep track of its position in real time. Vision-based localization (VBL) takes advantage of visual landmarks and image processing to extract information. Camera based solution is an efficient and cost-effective option since we're using our smart phone camera which has the requisite hardware, making the Vision Based Localization approach a beneficial choice for designing indoor navigation system.

For indoor navigation system based on Vision Based Localization we need to have a marker-based detection, for which we have two available options – fiducial marker or natural marker. Now natural marker tracking can be little difficult and often tricky because it depends on the object's natural features which can be different under different lighting conditions or circumstances. That is why we're making use of ArUco marker which is basically a Fiducial marker constructed with the sole purpose of encoding/decoding information in the marker and tracking the marker **[2]**.

1.2 ArUco markers (over other fiducial markers)

ArUco marker is a square binary fiducial marker that is commonly used in AR applications in order to determine position. The aruco module is based on the ArUco library, a popular library for detection of square fiducial markers developed by Rafael Muñoz and Sergio Garrido **[4]**.

Advantages of using ArUco marker: -

a) The ArUco library used for marker detection is very fast and optimized which makes it is a good choice for video capturing.
b) The ArUco markers provide information in encoded format without adding any overhead with unnecessary complexity.
c) We are also able to find out the heading information without any extra computation. It returns the top right, top left, bottom right and bottom left corners just by scanning the marker, thus from this information we can very easily find out the head position of the bot and relay instructions accordingly.
d) ArUco markers are made of a 2D array of black and white variations of which only a few are used for id generation with the rest being available for error detection. So it is even great for detecting marker that are occluded due to external factors **[3]**.

1.3 ArUco-Scanner application (over IP webcam and other methods)

A lot of things need to be taken care of for the bot to move in the right direction. So, we need an overhead camera that can translate the position of marker/bot to the client/PC as corner coordinates, then we need to calculate the centre and head position of the particular marker, compute the distance between the marker and the destination, and then send the signal through a message broker to the bot.

While all of this is time consuming process, the IP webcam which we had initially planned to go ahead with had latency issues thus adding to the problem. The problem was, IP webcam app had limited functionality i.e. it would translate the live video feed frame by frame to the client/PC, after which all calculation i.e. corner detection, centre and head calculation, distance computation between the marker and destination is done at the client/PC. This leads to a time delay and thus makes the whole process laggy and not effective in real-time.

Instead we chose ArUco-Scanner because it reduces the load on the client. It makes use of the smart phone's processing power to identify the corner coordinates of the marker and calculate the centre. Finally instead of sending live video feed to the server, like IP webcam, it forwards just the corner and centre coordinates to the server. This greatly reduces the latency and it appears as if everything is happening at real-time.

Previous work done and challenges imposed:

Sl. No.	Author Name	Paper Name	Contribution	Challenges
1	Babinec, Andrej & Jurišica, Ladislav & Hubinský, Peter & Duchoň, František	Visual Localization of Mobile Robot Using Artificial Markers[5]	ArUco tags lowered the cost of implementation of visual localization	Distance between camera and markers play a major role and reduce effectiveness and accuracy of using smaller ArUco markers
2	Alves, Paulo & Costelha, Hugo & Neves	Localization and navigation of a mobile robot in an office-like environment[6]	Global localization and ROS enabled navigation showed promising results	Hardware implementation involved webcam mounted robots which increase development costs

CHAPTER-2

System Design

2.1 Project Architecture

Our proposed system consists of the following four modules:

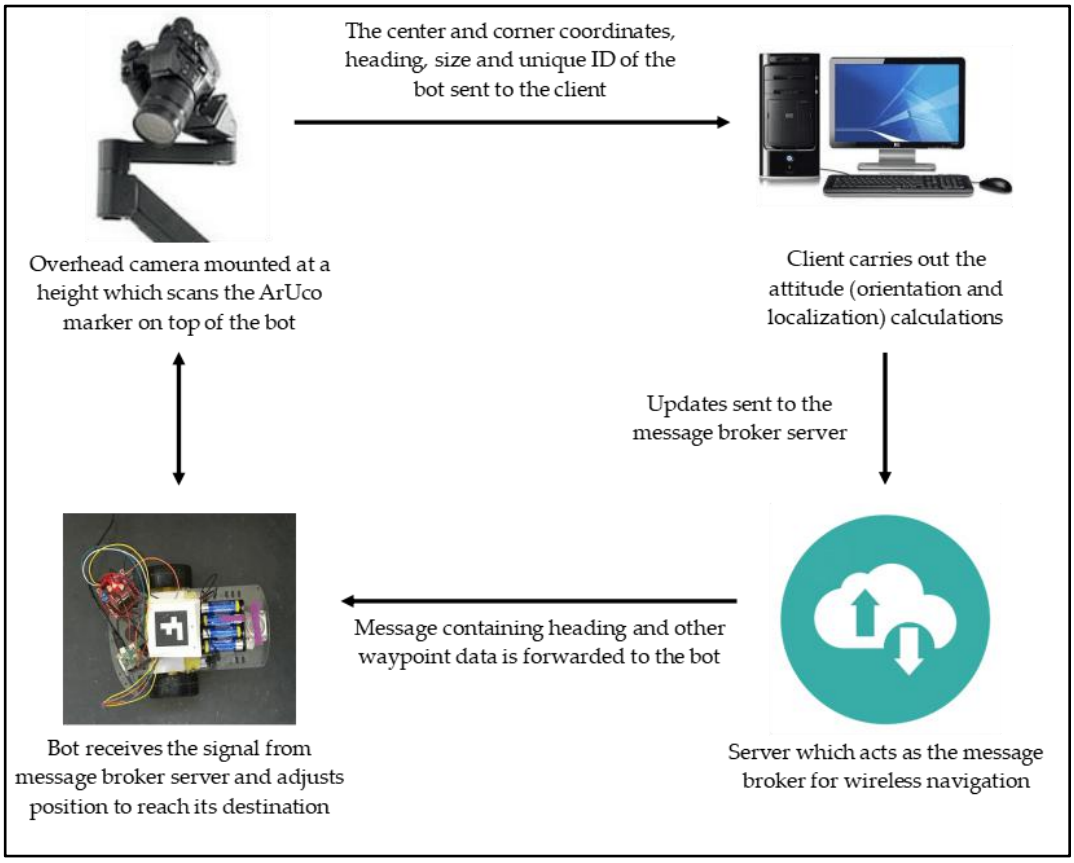

Fig. 2.1. High level abstraction of data and instruction flow among different interacting components within the project

The message broker protocol popularly used is MQTT (Message Queuing Telemetry Transport) which is a lightweight, publish-subscribe network protocol that transports messages between devices.

2.2 Project Work-flow

The following flowchart shows the detailed view of the processes happening at each step and how data and instructions are forwarded to the next entity after each subsequent step in the work-flow cycle for further processing.

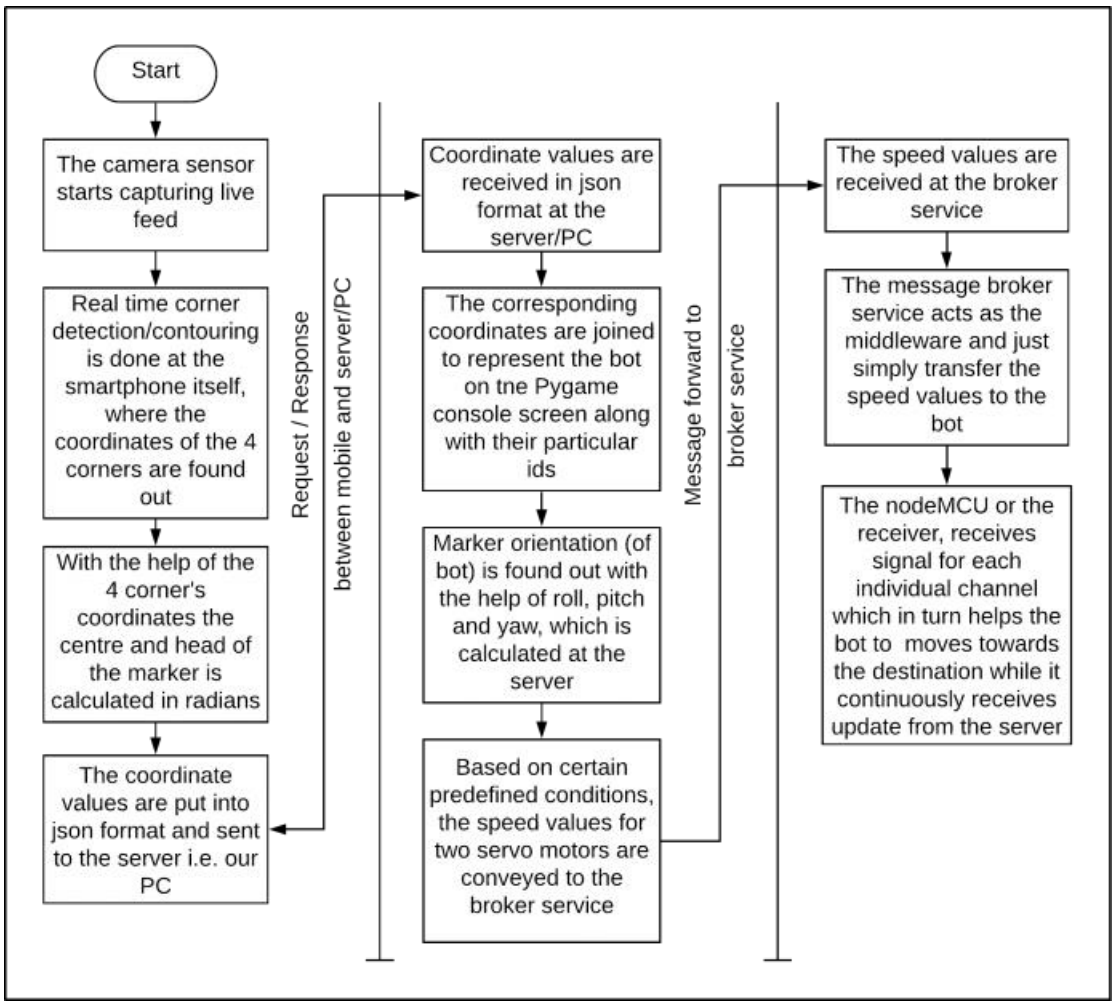

Fig. 2.2. Comprehensive view of work-flow and data exchange among different interacting components within the project

Methodologies of Implementation

This project primarily addresses the issue of indoor navigation using image processing which is implemented with the help of the OpenCV python module. In that regard, our chosen study of methods deals with the following implementation stages (with detailed algorithms and development specifications discussed later on).

The implementation of the ArUco-Nav project is achieved in the following stages -

1. Generation of ArUco markers
2. Camera calibration
3. Detection of ArUco markers -
    a) Using a standard webcam
    b) using IP webcam (Android application)
    c) using ArUco Scanner (Android application) [chosen method for this project]
4. Processing the video feed for obtaining marker orientation and simulating robot localization
5. Implement waypoint navigation between multiple targets by communicating with the swarm robots (using MQTT/TCP or other suitable protocols)

The ArUco markers used in this project have been generated using a web application that we have designed (using Bootstrap, JavaScript and jQuery). All markers belong to the original ArUco (5x5) dictionary.

OpenCV helps in successfully detecting the ArUco markers and identifying the center coordinates along with the corner values. The centre of the aruco marker is calculated at the video source and the data is relayed to the server which then calculates the distance between the x and y coordinates of ArUco marker and x and y coordinates of the target.

The main objective here is to make both the coordinates same i.e. the coordinates of an ArUco marker should overlap with that of the target. In this way we can make the bot traverse a predefined path autonomously without any human intervention.

Software and Hardware Requirements

2.3 Software Requirements

1. ArUco-Gen [web based ArUco marker generator]
2. ArUco Scanner application (for Android OS) [release used: v1.1.0]
3. Windows XP/7/10 with Python 3 installed [version used: Windows 10 64-bit with Python 3.8.2]
4. Message broker server [preferably MQTT]
5. Arduino IDE
6. Python 3 modules required:
    a) ar-markers==0.5.0
    b) numpy==1.18.3
    c) opencv-contrib-python==4.2.0.34
    d) opencv-python==4.2.0.34
    e) paho-mqtt==1.5.0
    f) simple-pid==0.2.4

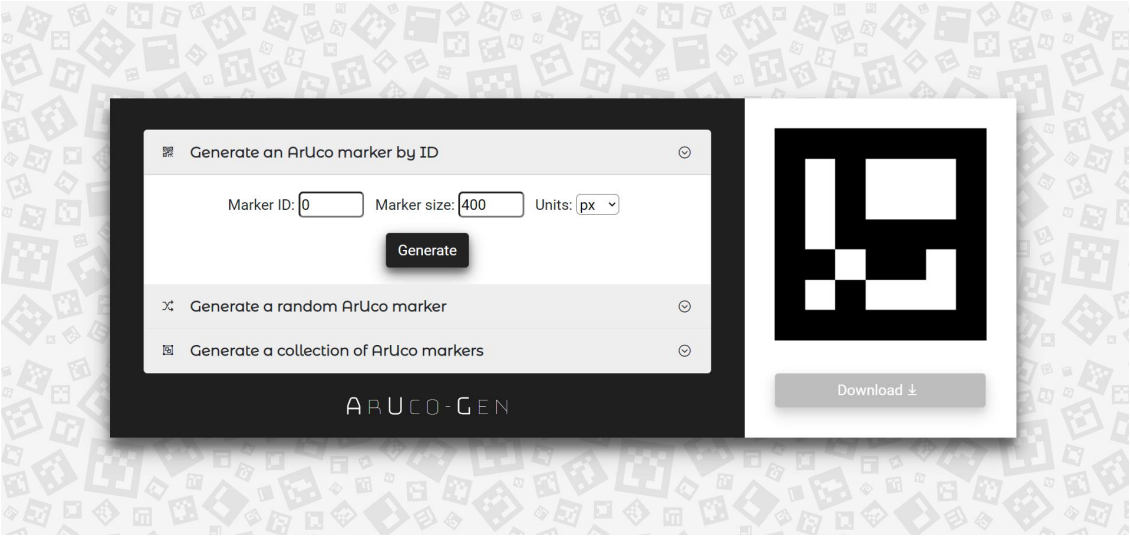

Fig. 2.3. ArUco-Gen web app for creating ArUco markers with fixed and random IDs

2.3 Hardware Requirements

1. Android smart phone with camera (at least 720p resolution)
2. Micro-mouse (or a standard 2WD robot)
3. Arduino Nano (for robot locomotion)
4. IoT development board (for wireless communication) [Bolt IoT]
5. PC/Laptop with minimum 2GB of RAM and Wi-Fi capabilities [system used: ASUS Vivobook S (X510U) with Intel Core i5-8250U x-64 based CPU and 8GB RAM]

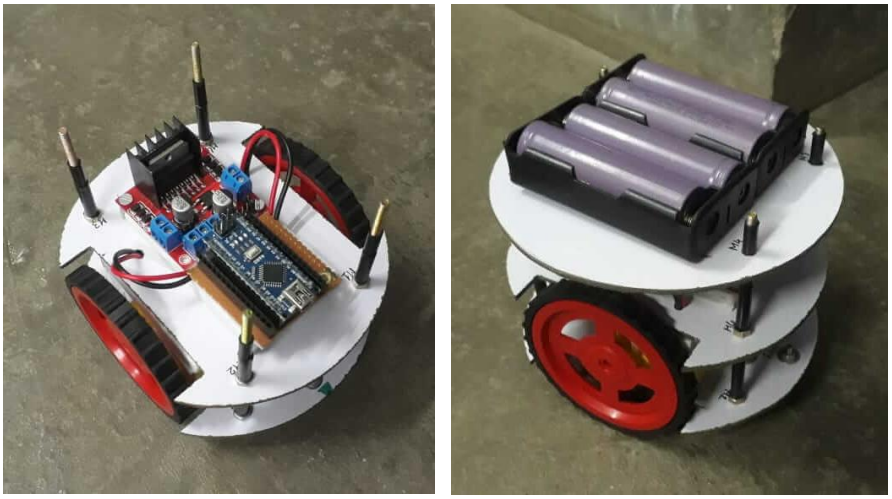

Fig. 2.4. 2WD robot chassis with Arduino Nano, L298N motor driver, BO-motors for drive and 3.7V 1500mAh Li-ion cells for powering the bot

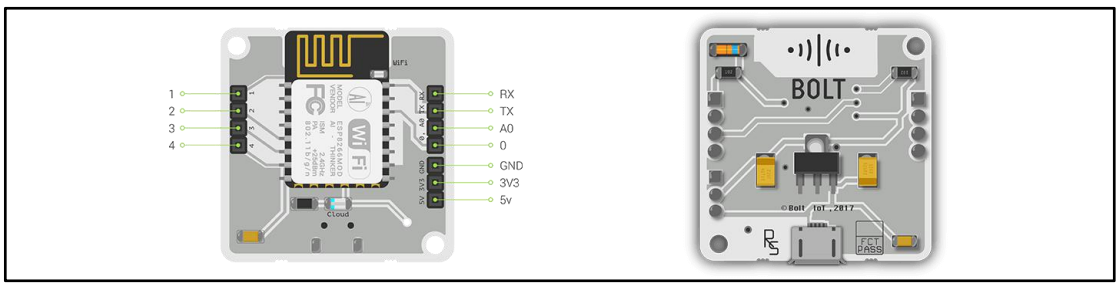

Fig. 2.5. Bolt IoT Wi-Fi module with ESP8266-12E

Our Approach

Our proposal to solve the localization and navigation problem for a robot (or a swarm of robots) is to use an overhead fiducial marker detector. Rather than using the overhead camera simply as a source of continuous video feed, we use the computing capabilities in modern day smart phones to process the video output and return marker data packaged with the response.

Our strategy to generate ArUco markers involved the use of a robust web-based marker generator that we designed using JavaScript. Instead of using simple library-based marker generators that output markers with several restrictions, we have developed an application that can generate markers having fixed or random IDs of any size.

Our approach to solve marker detection was to use an open-source project titled **Aruco Android Server** which we modified to include more specific computations pertaining to our project. The new application named **ArUco Scanner** not only scans for fiducial markers but also computes the marker ID, corner and center coordinates, heading (in radians) and the marker size.

Compared to other detection schemes like using a standard webcam or a popular Android application like IP webcam, ArUco Scanner achieves a lot more by sending a packed JSON response containing marker data which would otherwise consume unnecessary processing power on the client side.

A more detailed implementation flow has been discussed in the chapter below.

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.

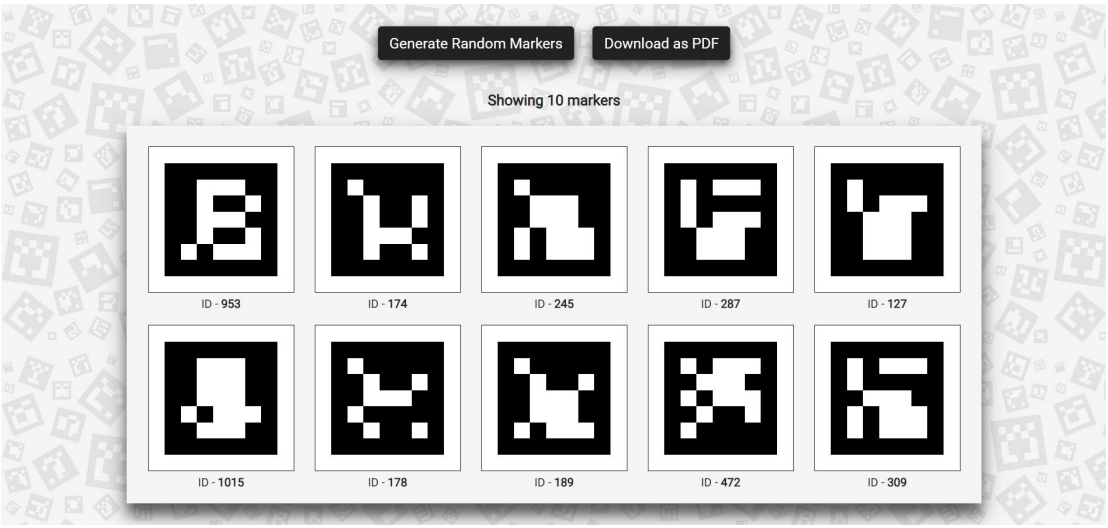

Fig. 3.3. Generating multiple ArUco markers at a time (up to 500 markers allowed)

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]

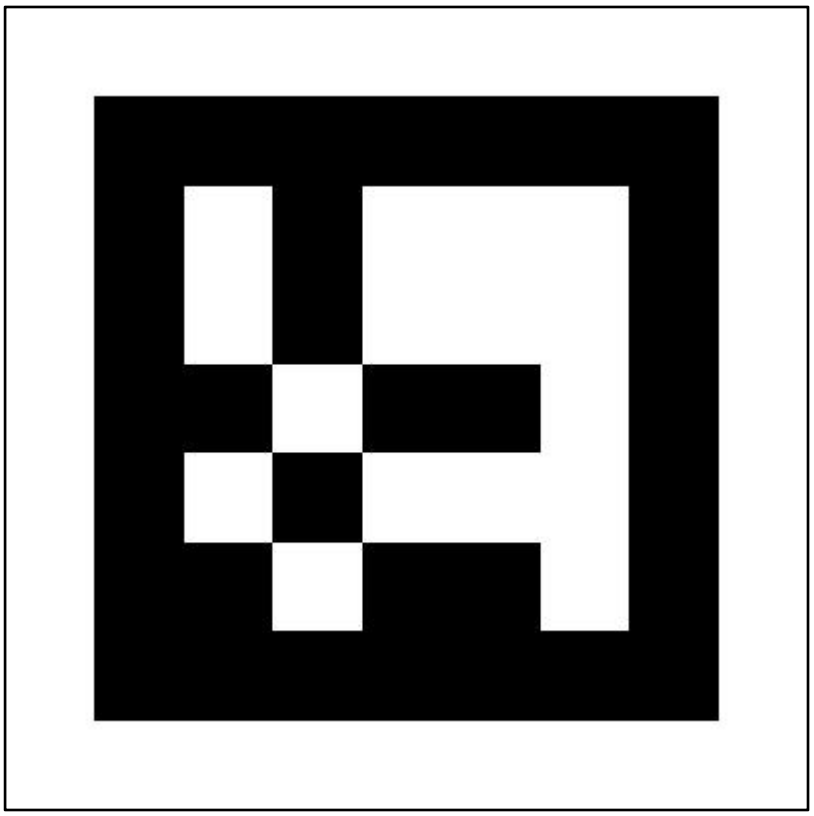

Fig. 3.4. An ArUco marker with an ID of 358 and size of 400px X 400px

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)

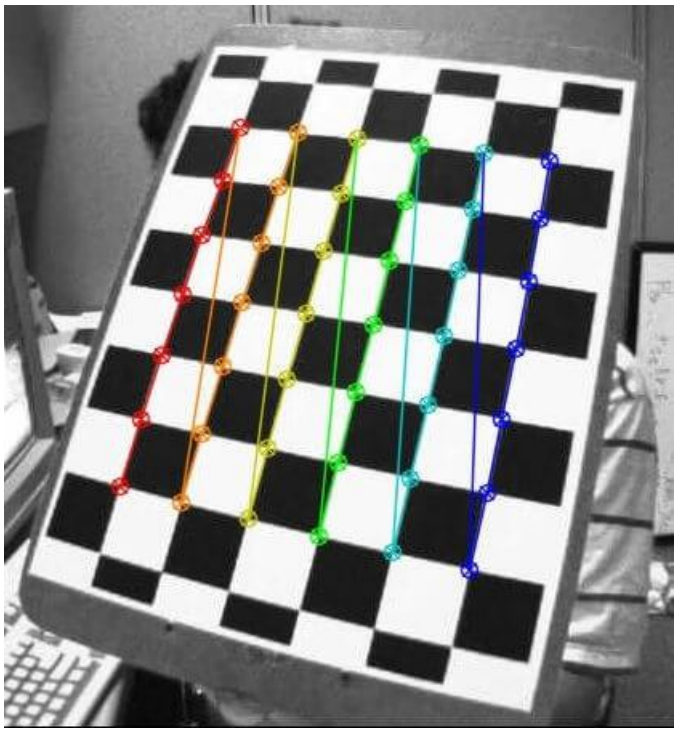

Fig. 3.6. 7x6 test pattern drawn from left-to-right, top-to-bottom

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)

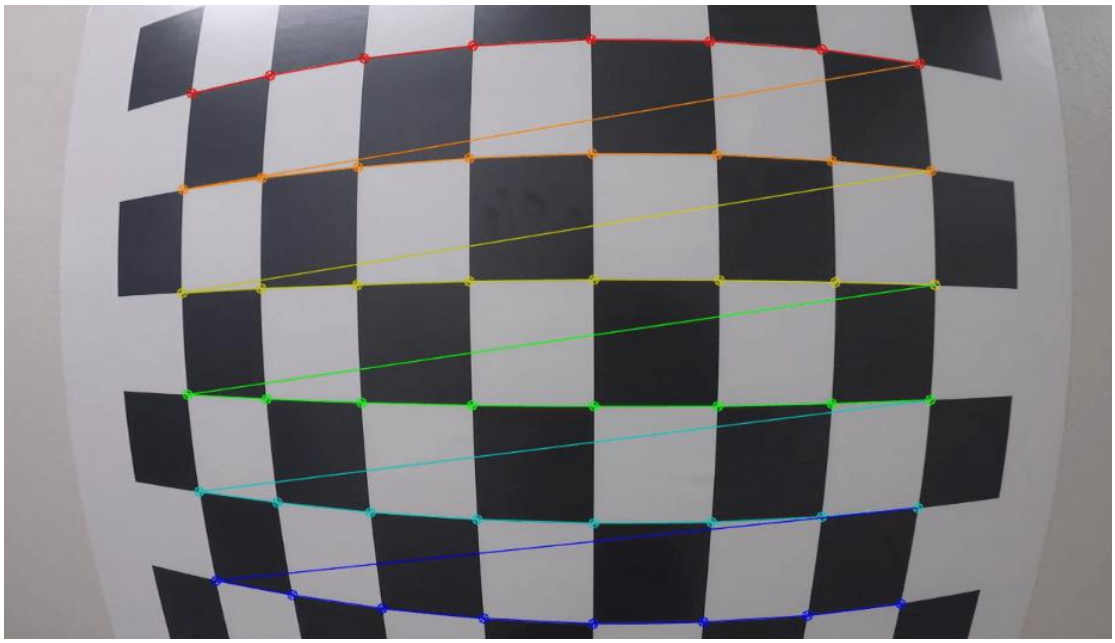

Fig. 3.7. Using either cv2.undistort() or remapping to undistort the image

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

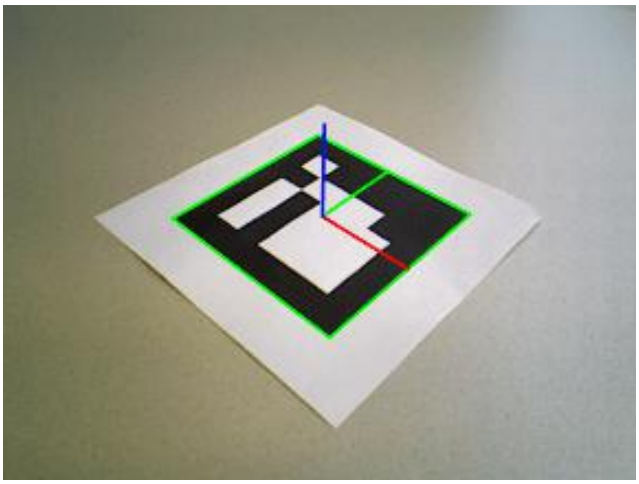

Fig. 3.8. An ArUco marker with a green bounding box and three axes drawn at the center (shown using three different colors - x: red, y: green, z: blue)

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].

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

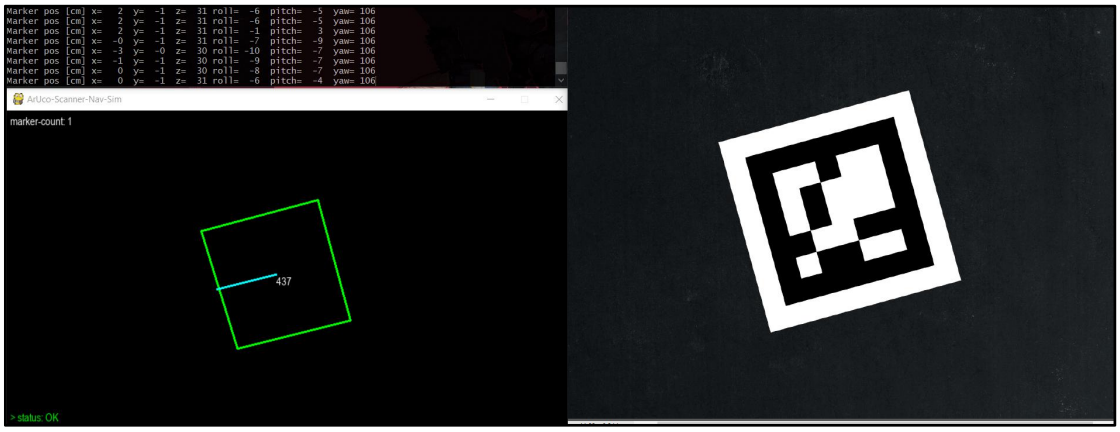

Fig. 3.11. Pose estimation of an ArUco marker

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.

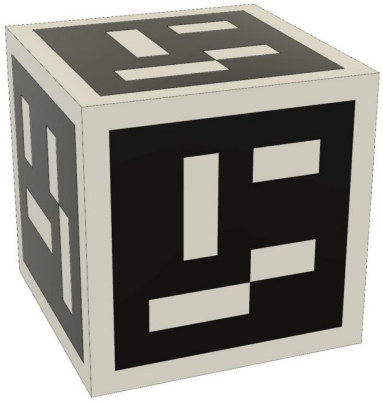

Fig. 3.12. An ArUco marker cube for a sample orientation test

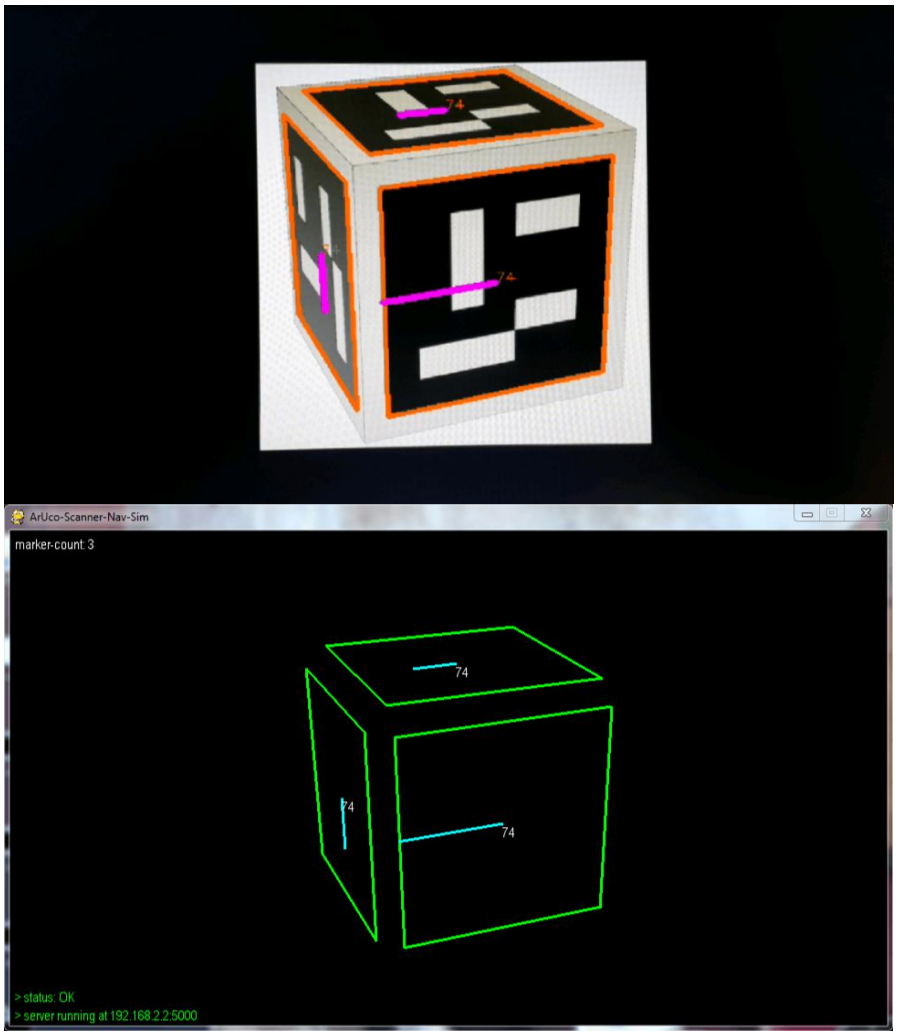

Fig. 3.13. Scanned output (top) and localization with distinct orientation (bottom)

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).

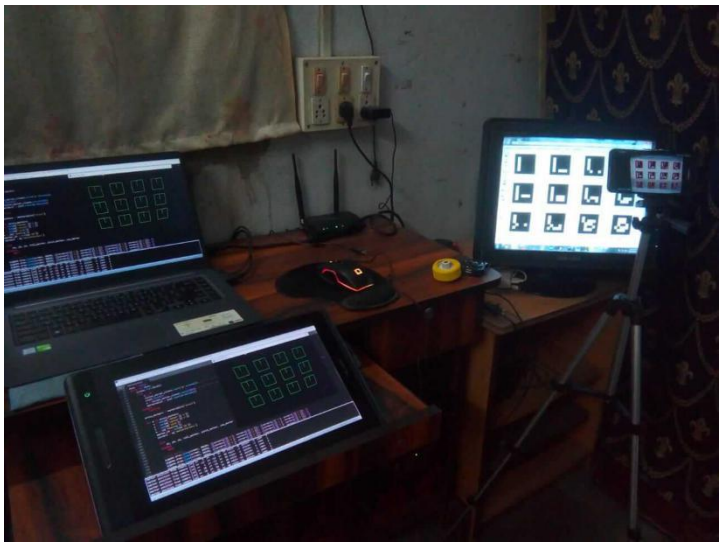

Fig. 3.14. ArUco marker localization setup (with ArUco Scanner running on an Android smart phone and multiple random ArUco markers displayed on a screen)

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).

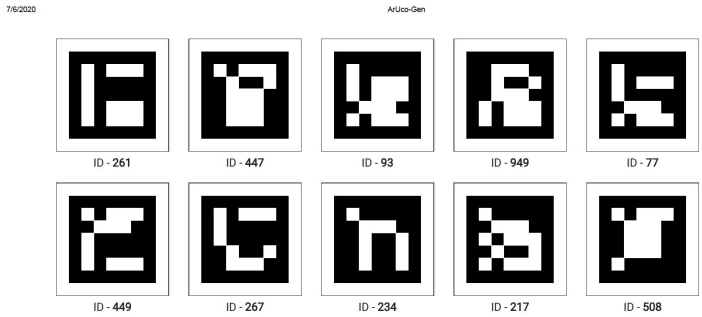

Fig. 3.15. Ten random ArUco markers generated using the ArUco-Gen web app

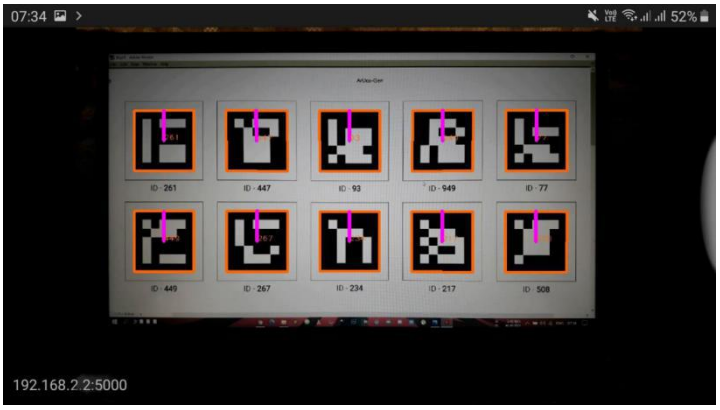

Fig. 3.16. Bounding boxes and heading lines generated for the ten markers using the ArUco Scanner Android application. The JSON response sent by the server to the client contains marker id, size, coordinates and heading (in radians)

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.

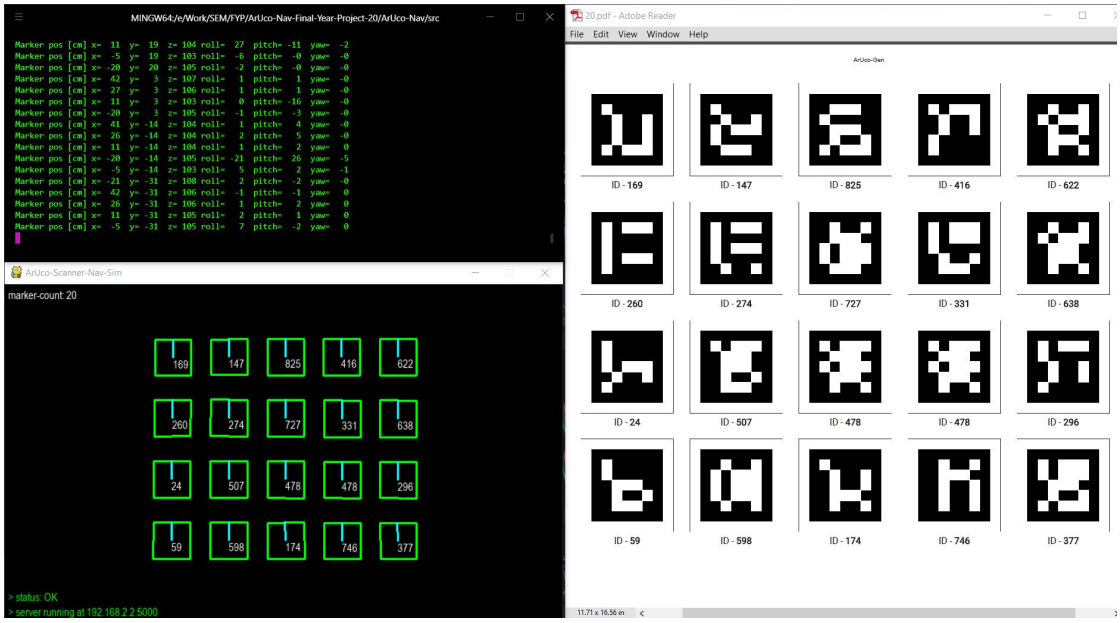

Fig. 3.18. A consolidated representation of the localization process - (from the right clockwise), a set of 20 ArUco markers, the marker orientation (attitude) data being printed on the terminal, the pygame window showing the positions of all 20 markers

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).

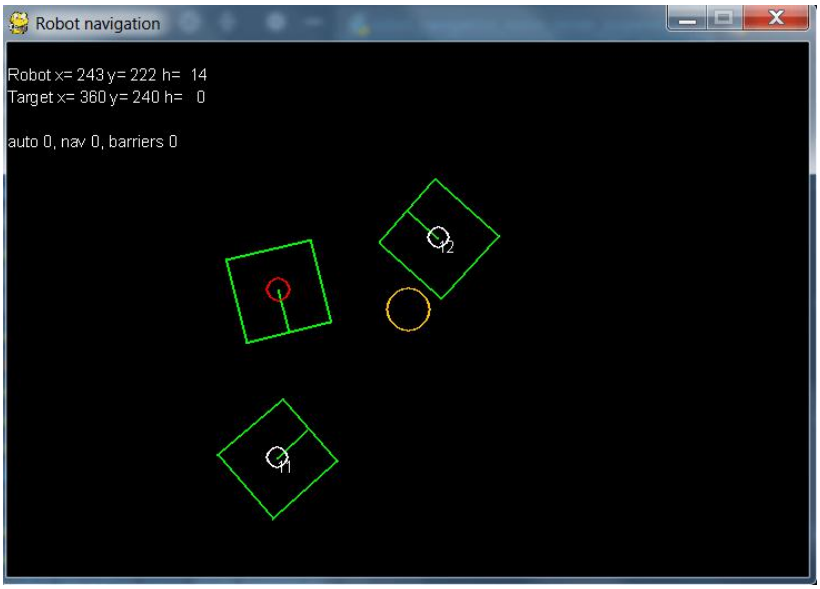

Fig. 3.19. A test simulation showing a robot (marked with a red circle at the center), two barrier objects (marker IDs 11 and 12) and a target (yellow circle)

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)

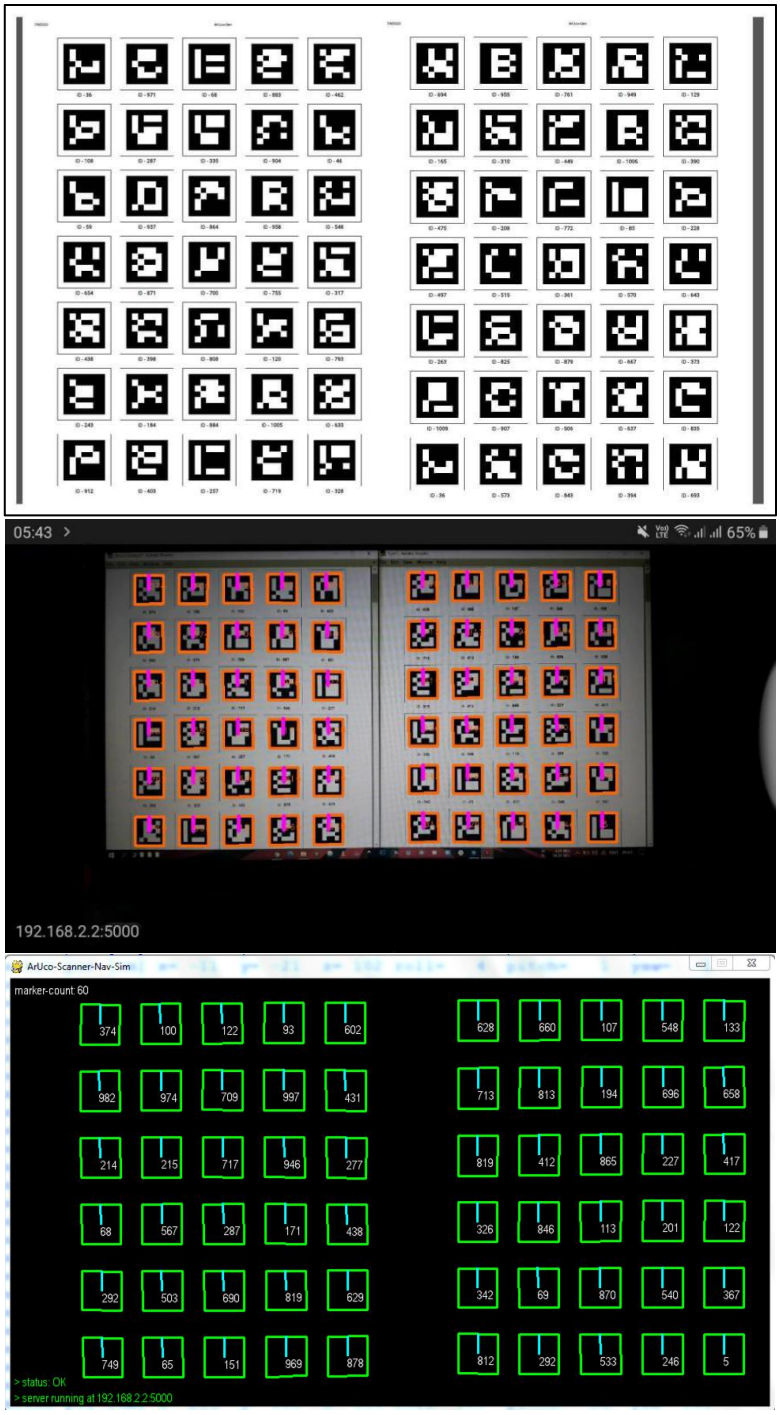

Fig. 3.21. System validation [from top, ArUco-Gen (generator), ArUco Scanner (detector) and ArUco Scanner Nav Sim (localization and navigation simulator)]

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).

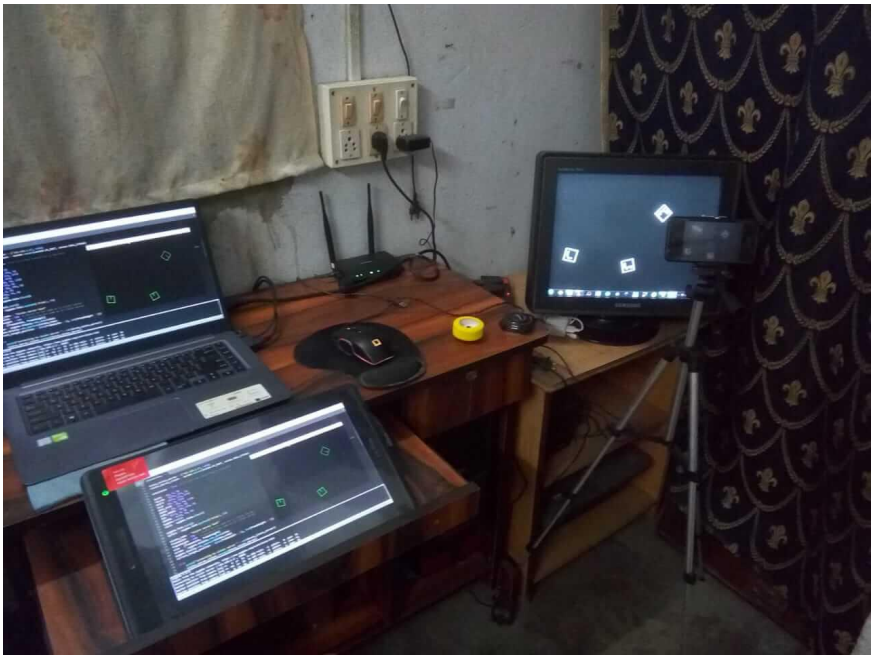

Fig. 3.22. A simple localization and navigation setup using ArUco Scanner on a smart phone mounted on a tripod and the simulation logged on the laptop screen

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.]

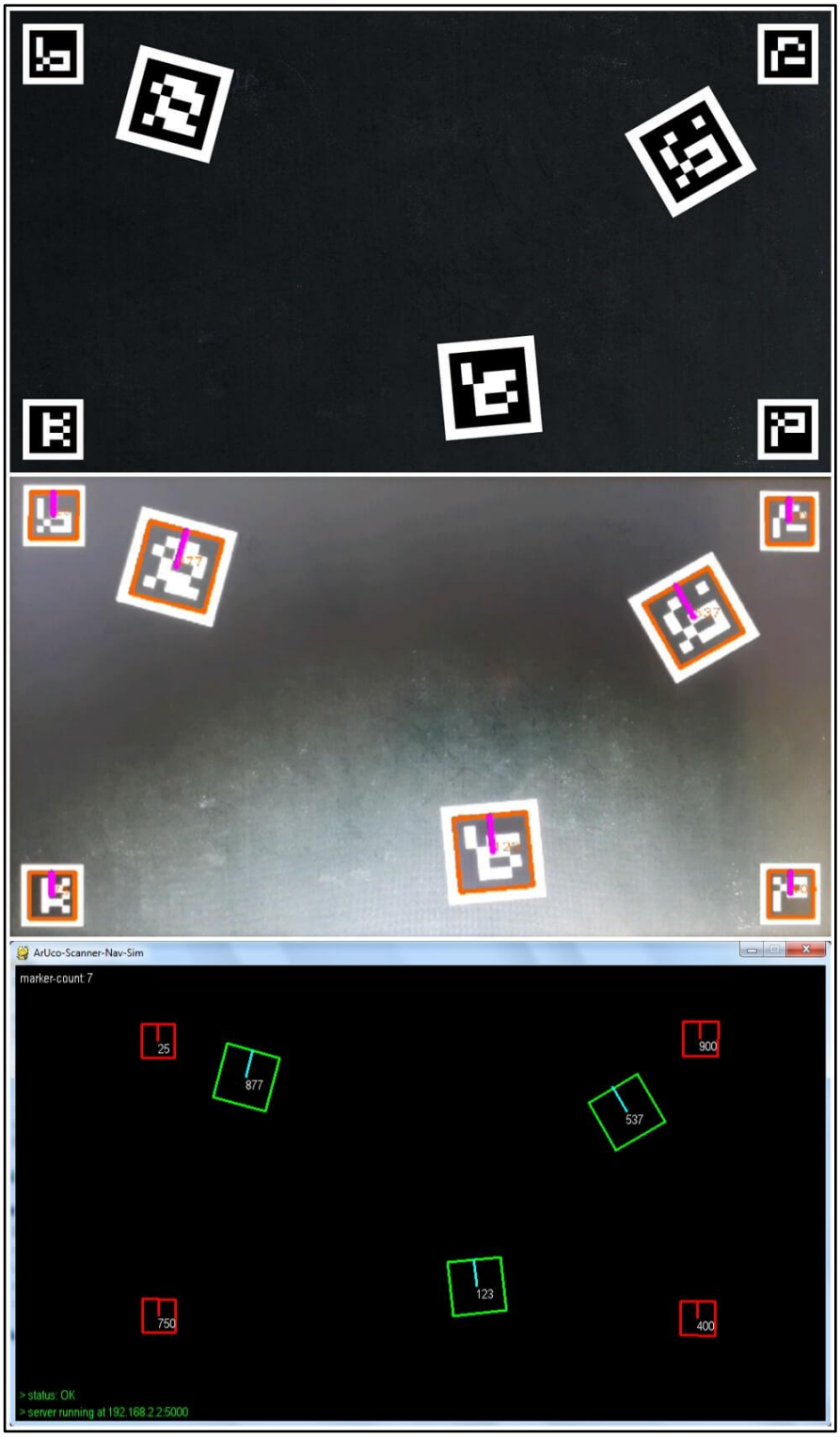

Fig. 3.23. output - time-stamp: 00:01:25; frame no.: 5

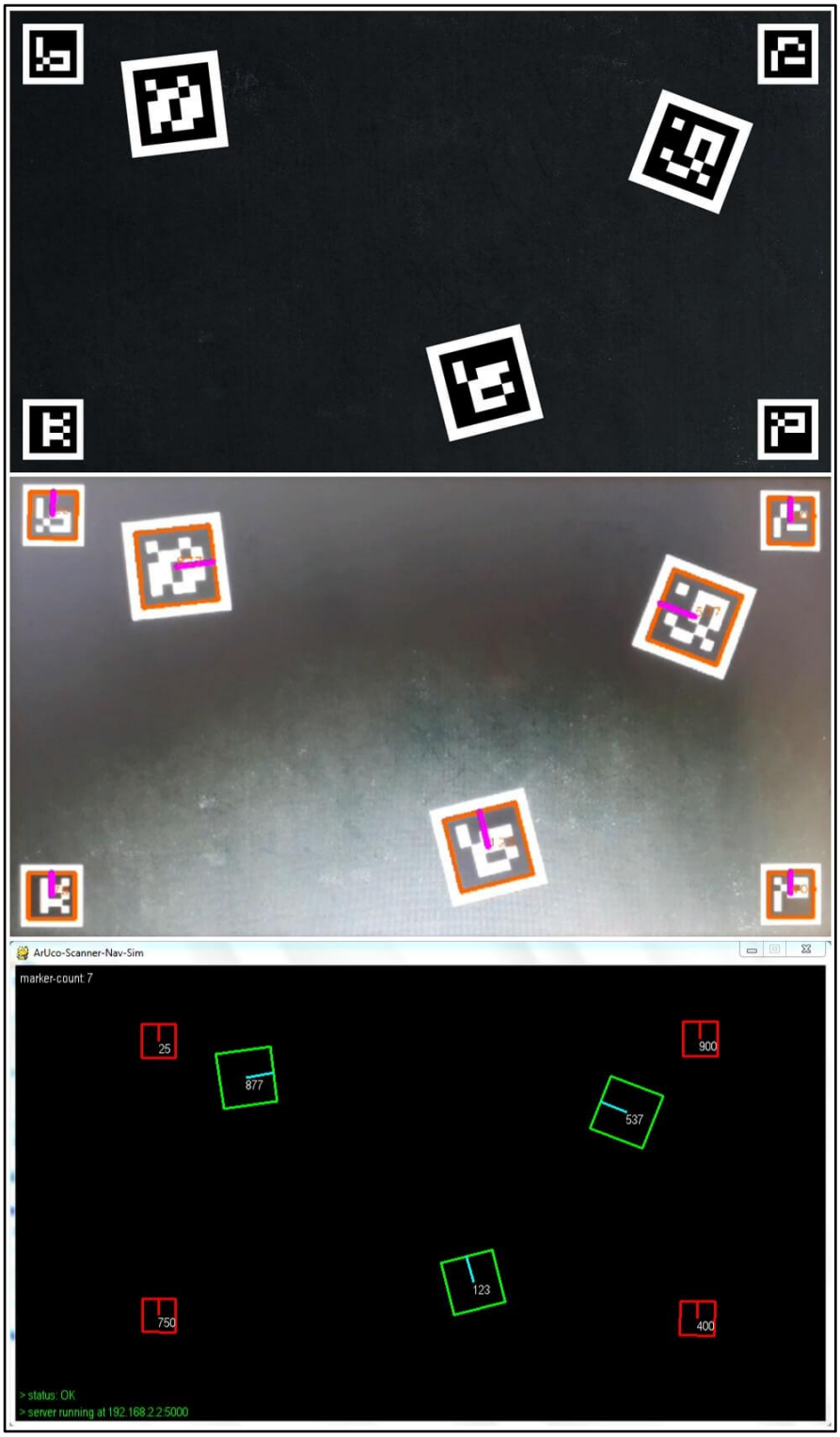

Fig. 3.24. output - time-stamp: 00:03:75; frame no.: 15

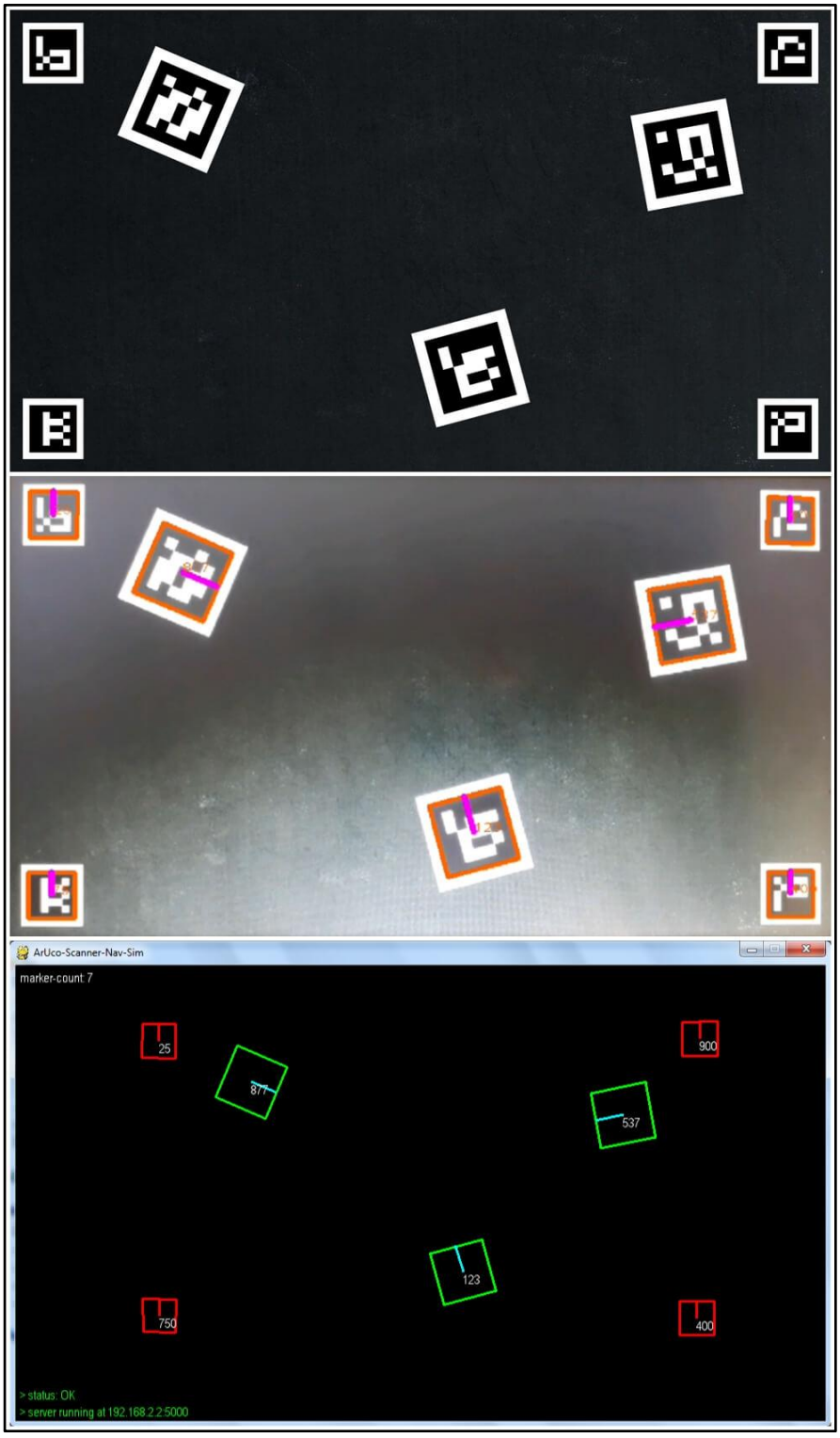

Fig. 3.25. output - time-stamp: 00:06:25; frame no.: 25

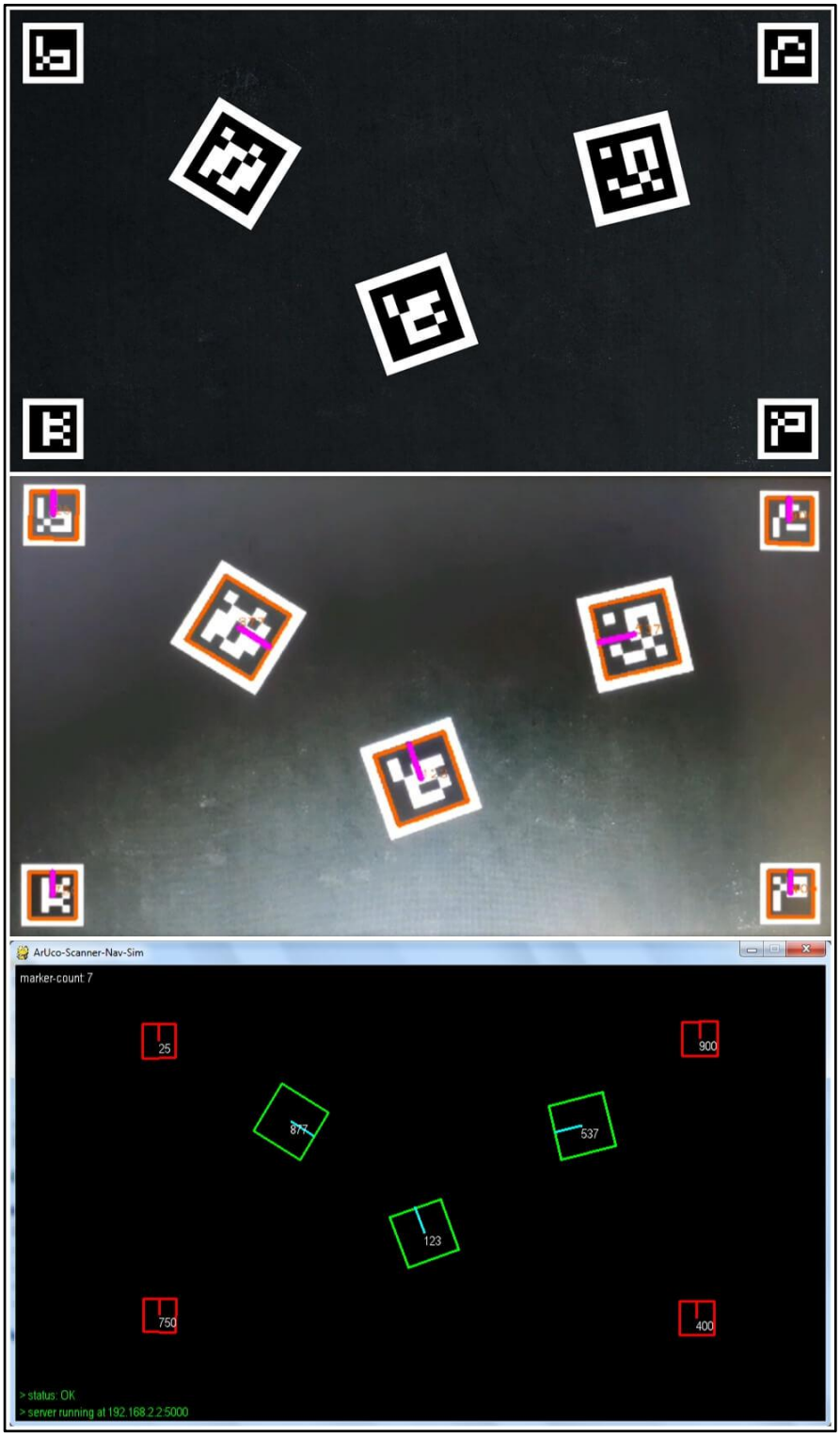

Fig. 3.26. output - time-stamp: 00:11:25; frame no.: 45

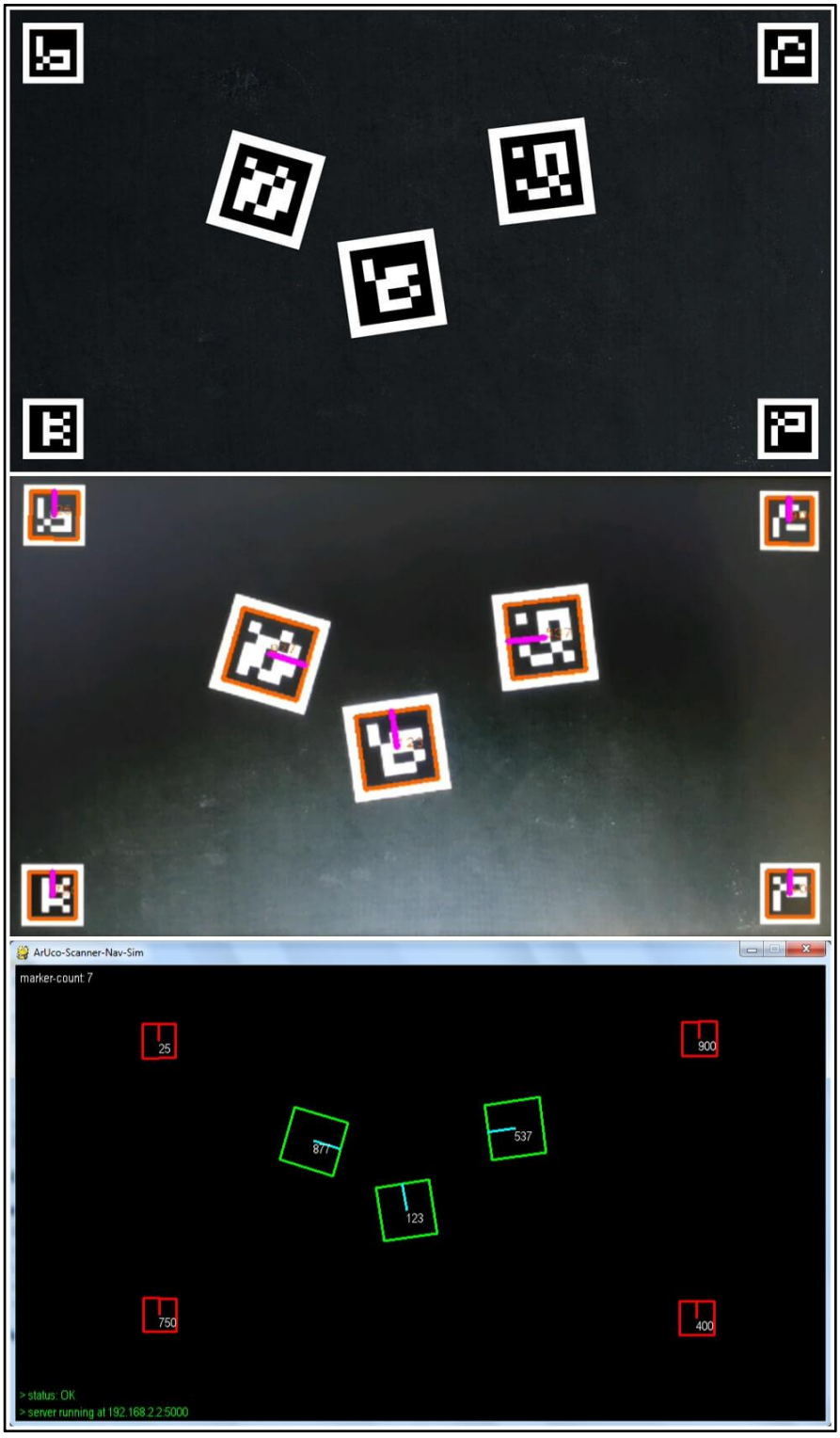

Fig. 3.27. output - time-stamp: 00:13:75; frame no.: 55

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.

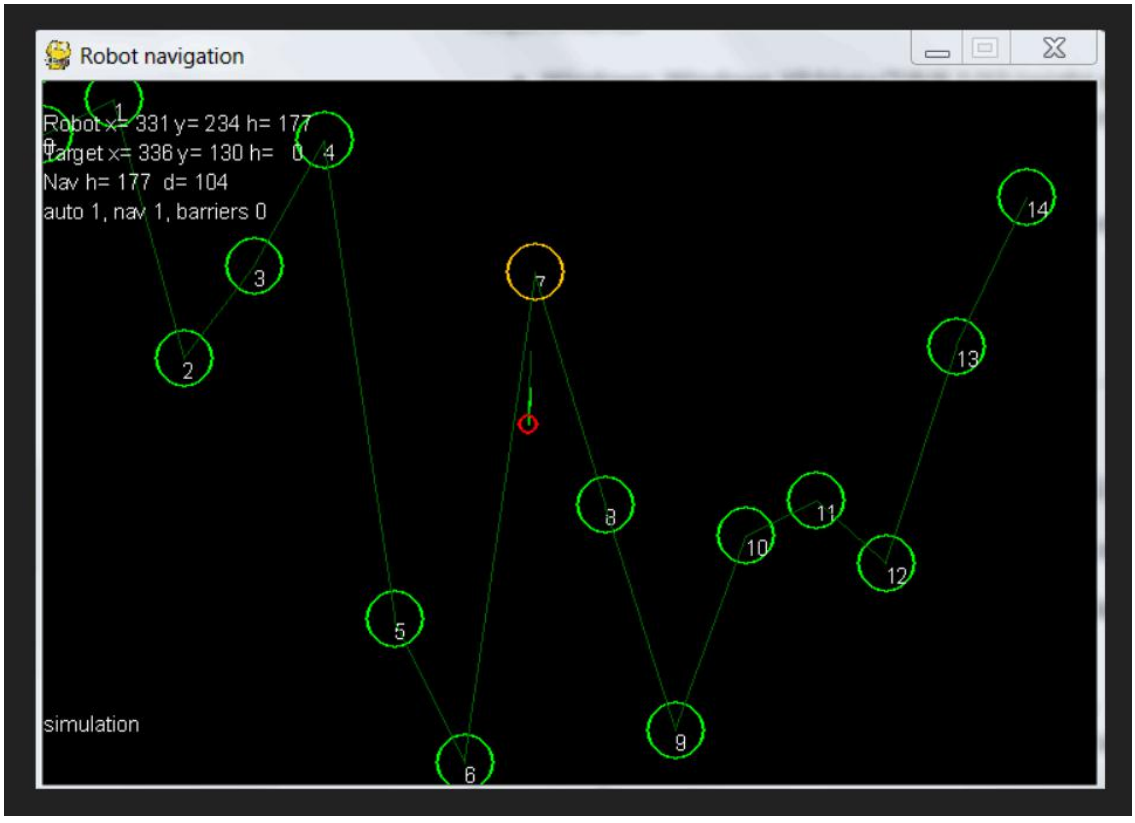

Fig. 3.28. Intended simulation of way-point navigation of a swarm of robots with barriers and dynamic way-points.

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

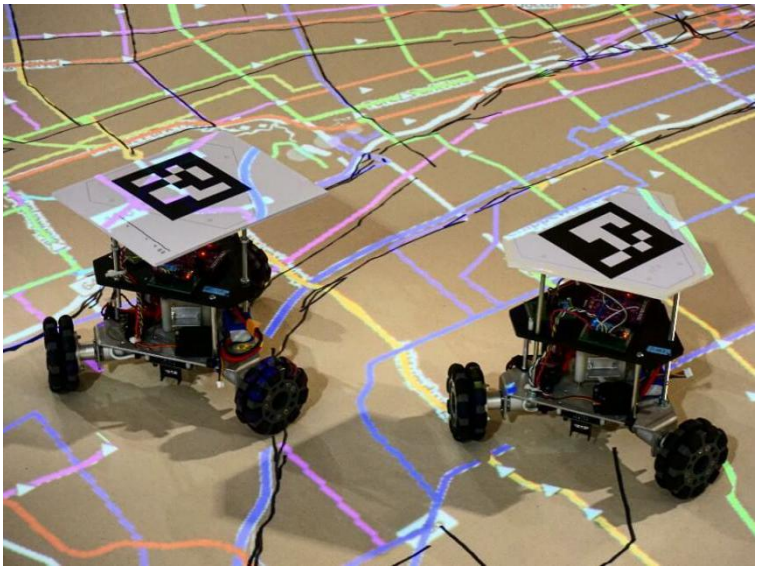

Fig. 3.30. A swarm robot configuration with ArUco markers mounted on each robot

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)

Appendix - Source Code for the ArUco-Nav project

Project structure

```
ArUco-Nav
  |__ camera_calibration_config
  |       |__ cameraMatrix.txt
  |       |__ distortionCoefficients.txt
  |__ src
  |       |__ aruco_nav_sim.py
  |       |__ marker_orientation.py
  |__ .gitignore
  |__ README.md
```

--camera_calibration_config/

1. cameraMatrix.txt

5.09623562867339217e+02,0.00000000000000000e+00,3.24752342861335307e+02
0.00000000000000000e+00,5.08729992602668744e+02,2.47943091195423249e+02
0.00000000000000000e+00,0.00000000000000000e+00,1.00000000000000000e+00

2. distortionCoefficients.txt

1.61026040362261924e-01,-2.52877605985532050e-01,1.88978194588328650e-03,
-7.59855731192305198e-04,-2.49835849286275640e-01

-- src/

3. aruco_nav_sim.py

```
# importing required modules
import math
import json
import socket
import pygame
from marker_orientation import getMarkerOrientation

# function to display text on the pygame window
def blitText(str, color, x, y):
   text = myFont.render(str, True, color)
   screen.blit(text, (x, y))
```

```
# storing server url
aruco_vision_server_IP = "192.168.2.2"
port = 5000
aruco_vision_server = (aruco_vision_server_IP, port)
 # creating a socket instance
aruco_vision_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# pygame window name
window_name = "ArUco-Scanner-Nav-Sim"
# pygame window size
width = 1280
height = 720
size = [width, height]
# pygame colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
CYAN = (0, 255, 255)
GREEN = (0, 255, 0)

# initializing pygame
pygame.init()
pygame.font.init()
# creating time and font objects
clock = pygame.time.Clock()
myFont = pygame.font.SysFont("Arial", 15)
# setting up pygame window
screen = pygame.display.set_mode(size)
pygame.display.set_caption(window_name)

# establishing connection to ArUco-Scanner vision server
try:
  print("Connecting to ArUco-Scanner vision server " + str(aruco_vision_server))
  aruco_vision_client.connect(aruco_vision_server)
  aruco_vision_client.setblocking(0)
  aruco_vision_client.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY,
1)
  aruco_vision_client.settimeout(1)
except:
  print("Unable to connect to ArUco-Scanner vision server" + str(aruco_vision_se
rver))
   exit()
```

```
done = False
while not done:
    # setting background color to black
    screen.fill(BLACK)

    # getting scanned markers
    try:
        s = ""
        # sending request to ArUco-Scanner vision server
        aruco_vision_client.send("g".encode())
        clock.tick(5)
        # storing JSON response containing marker data
        s = aruco_vision_client.recv(100000)
        # populating marker dictionary with received marker data
        markersDict = json.loads(s.decode())
    except:
        continue

    # storing JSON data for all received markers
    aruco_markers = markersDict["aruco"]

    # showing connection status and marker count info
    connection = "> server running at " + aruco_vision_server_IP + ":" + str(port)
    count = "marker-count: " + str(len(aruco_markers))
    status = "> status: OK"
    blitText(count, WHITE, 5, 5)
    blitText(status, GREEN, 5, (height - 40))
    blitText(connection, GREEN, 5, (height - 20))

    # getting individual marker specs
    for m in aruco_markers:
        # getting marker ID
        marker_id = int(m["ID"])
        # getting marker size
        marker_size = int(m["size"])
        # getting marker heading direction in radians
        marker_heading = m["heading"]
        # getting corner coordinates
        corners = m["markerCorners"]
        # getting center coordinates
        Xc = int(m["center"]["x"])
        Yc = int(m["center"]["y"])

        # computing center coordinates of the top side of the marker
        Xt = int(Xc + ((marker_size / 2) * math.sin(marker_heading)))
        Yt = int(Yc + ((marker_size / 2) * math.cos(marker_heading)))
```

```
            # displaying marker attitude parameters
             try:
                xm, ym, zm, roll_marker, pitch_marker, yaw_marker =
            getMarkerOrientation(corners, 10)
              except:
                pass

            # drawing markers on the screen
            # drawing bounding box
            for i in range(4):
                j = ((i + 1) % 4)
                # start position coordinates
                Pi = (int(corners[i]["x"]), int(corners[i]["y"]))
                # end position coordinates
                Pj = (int(corners[j]["x"]), int(corners[j]["y"]))
                # line connecting start and end positions
                if marker_id in [25, 400, 750, 900]:
                 # barrier markers
                    pygame.draw.line(screen, RED, Pi, Pj, 3)
                else:
                    pygame.draw.line(screen, GREEN, Pi, Pj, 3)

            # drawing heading line
            if marker_id in [25, 400, 750, 900]:
             # barrier markers
                pygame.draw.line(screen, RED, (Xc, Yc), (Xt, Yt), 3)
            else:
                pygame.draw.line(screen, CYAN, (Xc, Yc), (Xt, Yt), 3)
            # showing marker ID at the center
            blitText(str(marker_id), WHITE, Xc, Yc)

        # updating the full display
        pygame.display.flip()

        # quitting event loop
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                done = True

    # exiting pygame
    pygame.quit()
```