Aerial Robotics Kharagpur Software Task Round 2022 Documentation

Upayan Chatterjee

***Abstract*— This documentation contains details about my solutions for the Task round of ARK Software Task round 2022**

For Task 2.1.1 and 2.1.2(Dollars and cents) I did familiarise myself with ROS framework and how nodes communicate and stuffs. However I could not materialise the code. I did make an attempt at the problem which I have enclosed.

For Task 2.2(Stereo Image based obstacle avoidance) I calculated the disparity map between the two images, located the bike in the image using `cv2.matchTemplate` and `cv2.minMaxLoc()`. Then using the disparity map and projection matrices, I generated the depth map. Finally using the depth map and obstacle(bike) location, I calculated the distance of the bike from the camera.

In Task 3.1(Detection and pose estimation), for part 1, I used the built-in `cv2.aruco` library to detect the aruco marker, show its id and estimate its relative pose using the `detectMarkers` and `estimatePoseSingleMarkers` functions. For part 2, I made a dictionary which stores the signature of a given aruco marker in all of the four orientations possible. Then, from the video frames, the program identifies all squares, perspective transforms them and determines the signature of the marker. If the signature is present in the predefined dictionary, then it is a valid aruco marker, otherwise not. For the pose estimation, I used the same signature dictionary, in which I also appended the "world" location of the four corners, and finally drew the axes on the video frames using the `cv2.line()` function. I used `cv2.solvePnP()` function to establish correspondence between the corners detected in the frame and the "world" locations stored in the aforementioned dictionary.

I. INTRODUCTION

TASK 2.1: The task demanded us to code an optimum strategy to a number guessing game. We had to make a guess number and the `centsdollar1` node would tell us how many digits from our guess were in the same position as in the original number, inferring from which we would have to make another guess number. The problem statement asked us to maximize our score which in turn implied to minimize the number of guess attempts. I unfortunately could not come up with a working code to this problem; however I have described my logic in the INITIAL ATTEMPTS segment.

TASK 2.2: This task was relatively straightforward in the sense that once one understands the underlying logic and mathematics, the code begins to simplify.

TASK 3.1: In this task the PART 1 required us to detect, identify and determine pose of an ArUco tag using inbuilt methods of OpenCV library. The PART 2 was the challenging part where we had to devise our own functions to do the same task. I have discussed my logic in the *Abstract* segment and more has been discussed in the INITIAL ATTEMPTS segment.

II. PROBLEM STATEMENT

TASK 2.1: *You are provided with a sequence of integers of length n. You can make guesses on what that number can be. Once you make a guess in return you will receive 2 outputs i.e. number of 'Cents' and 'Dollars'.'Cents' denote the number of digits guessed which are present in the given sequence but are out of position, whereas 'Dollars' represent the number of digits guessed which are present in the sequence at the very position you guessed them at. As you might have guessed it, the game ends when you get exactly n Dollars. Your solution must be optimised enough to run for strings of length 10 without repetition.*

TASK 2.2: *In order to find distances to obstacles we need depth information, however sometimes we don't have depth cameras so we end up using two cameras in specific orientation relative to each other to generate slightly different images called stereo images. In this task you are provided with a pair of stereo images. You need to generate a depth map, find the obstacle and calculate the distance. The obstacle is a blue bike which you have to avoid crashing into.*

Fig. 1. Image from left camera

Fig. 2. Image from right camera

Fig. 3. Bike: Obstacle

TASK 3.1.1: *In this task you have to take a picture of the ArUco marker given below and estimate its relative pose. You need to print the pose and tag id on the image. You can use inbuilt functions to find the pose of the ArUco tag, however you should be able to tell how they work, its limitations and the different parameters it takes up and how it affects its working. You also need to detect and estimate the pose using live video capture from your webcam at different distances.*

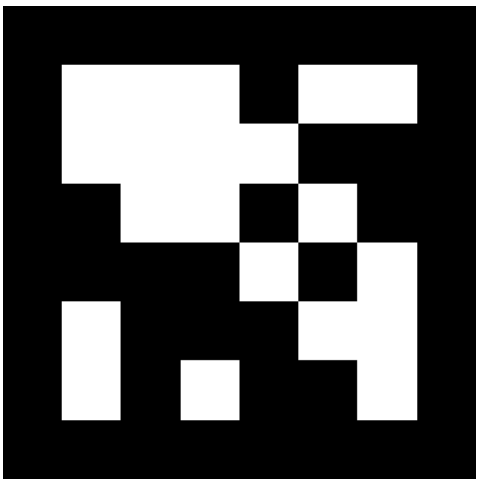

Fig. 4. ArUco Tag: 64

TASK 3.1.2: *As you now have the knowledge of detecting ArUco tags using inbuilt functions we want you to detect the tag using your own functions. You can use a number of methods to detect the ArUco tag. Not using inbuilt functions for pose estimation will fetch you extra points.*

III. Related Work

TASK 2.1: I started with the installation of `ros-noetic-desktop-full` on my Ubuntu 20.04. I learnt about ROS commands and about publisher and subscriber nodes. My understanding of the communication between the nodes is shown in Fig. 5. The 'player1' node publishes its guess to topic: '/guess_part1' which is subscribed to by the 'centsdollars1' node. The centsdollars1 node in turn publishes the number of cents and dollars to '/check1' topic which is subscribed to by the 'player1' node. It is thus a closed loop feedback system.

TASK 2.2: I started by reading about stereo vision and depth detection using stereo vision, while trying to understand the underlying mathematics behind the process of depth detection.

TASK 3.1: Initially during contour finding, I needed to threshold the image. However I was having a hard time in deciding the correct type of thresholding option. It was only later that I needed to use adaptive thresholding instead of global thresholding. The problem with global thresholding was that even slight shadows would make the marker completely black, and if I reduce the threshold value, the marker would turn completely white almost.

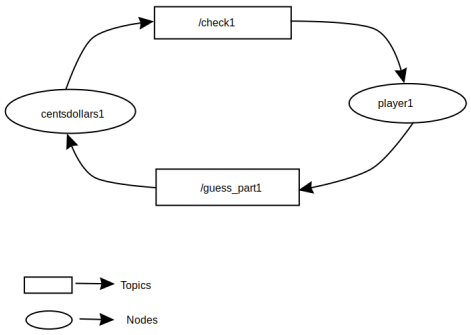

Fig. 5. rqt-graph

IV. Initial Attempts

TASK 2.1: I describe here the algorithm I had thought would be able to guess the correct number(which unfortunately could not be realised in the form of actual code):
Let 7923564180 be the number to be guessed.
step-1: First guess (`first_guess`) would always be 1234567890
Thus dollarcent would be 2
step-2: We keep on swapping the first digit of `first_guess` with all the remaining digits until we get a change in dollarcent(ΔD) as +1. In the meantime, if after some swap we get ΔD=-1, it would confirm that the number with which the first digit was swapped, was indeed in the correct position in the guess. Once the correct position of a digit has been ascertained, it is put in the appropriate index of an array (`num[]`) and not considered

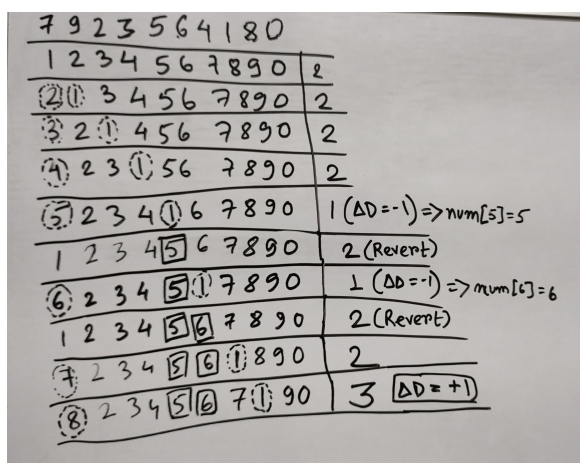

Fig. 6. Enumerating the steps(the second column of the table denotes the number of dollars and ΔD denotes the change in dollar value)

for any further swap operations.
step-3: When ΔD is +1, it implies that either of the numbers interchanged went into the right place(in this case either 1 has been put in the right place or 8 has been put in the right place after swapping). Now to determine which is the case, we change the `first_guess` to this swapped value. Then we swap the first digit of `first_guess` with the second digit- if even then ΔD ≥0 then it implies that 1 was the digit that was put in the right place after the previous swap. We thus store 1 in the appropriate index of `num[]`.
step-3.1: If instead of ΔD=+1 we get ΔD =+2 then it implies that after swap, both the numbers have gone to their proper place as in the original number and we store them in the `num[]`.
step-4: If dollarcent = 999, terminate, otherwise goto **step-2**.
N.B.- *It is to be noted that since the program was asked to be written for 10 digit numbers without repetition, the number of cents = 10 - number of dollars, thus only accounting for change in dollar is sufficient in this case.*

TASK 3.1: I read in an article that calibrating the camera was the first step in the detecting and identifying ArUco tags. Calibration of a camera basically gives us two matrices: camera matrix and distortion coefficients. Without calibration, certain distortions(pincushion or barrel distortions to be specific) creep into the images. Thus

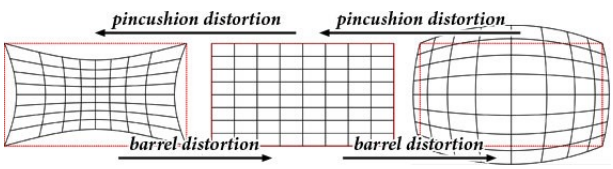

Fig. 7. Distortions

my first step was also to calibrate my camera using built-in opencv functions and a checkerboard pattern. I used the `cv2.findChessboardCorners()`, `cv2.cornerSubPix()` and `cv2.calibrateCamera()` to determine the camera_matrix and distortion_coefficients.

V. Final Approach

TASK 2.2: To determine the location of the bike in the picture, I used the `cv2.matchTemplate()` on the left image. I calculated the disparity map between the two images using the `cv2.StereoSGBM_create()`. Henceforth, I calculated the depth of the obstacle using:
$\mathrm{d} = f_{length} * \mathrm{baseline/disparity}$
The final distance came out to be 9.127 metres as shown in the OBSERVATION AND RESULTS section.

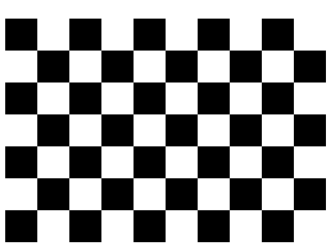

Fig. 8. Checkerboard pattern used to calibrate the camera

TASK 3.1:
For Part 1 I imported `cv2.aruco` *as* `aruco`. I converted all frames to grayscale(using `cv2.cvtColor()`) before working on them. After that I fetched the inbuilt 6X6 aruco dictionary using the `aruco.Dictionary_get()`, following which I invoked the `aruco.detectMarkers()` with parameters as: grayscale frame, `aruco_dict` and `aruco.DetectorParameters_create()`. Finally, to estimate pose, I utilised the `aruco.estimatePoseSingleMarkers()` function.

For Part 2 I followed the following steps:
Step-1: Using all four orientations of the ArUco marker(not from the video frames), create a dictionary which contains signature of the marker for each orientation(signature of a marker is nothing but a list of 1's and 0's corresponding to a white or black square respectively in the marker starting from the top left corner). In the dictionary, along with the signature, store the world coordinates of the corners of the marker also(let me explain what I mean by world coordinates: in case of an upright aruco marker, the top left corner of the marker is also the top left corner with respect to the world, however, when we rotate the marker clockwise through 90 degrees, the top left corner with respect to the marker is now the top right corner with respect to the world)
Step-1a: In order to determine the signature of the marker, we project the image to a 64X64 grid (such that each block of the marker would be 8X8 pixels in size), OTSU threshold the image and erode the image. Then from each 8X8 block we sample the colour at the centre of the block and push 1 in the list if it is white, else push 0.
Step-2: Convert all frames of video to grayscale.
Step-3: Apply adaptive thresholding to frames before detecting contours.
Step-4: Detect contours.
Step-5: Approximate contours using `cv.approxPolyDP()`.
Step-6: Reject those contours which do not have exactly 4 sides and those whose area is less than 100(worked in my case).
Step-7: We perspective transform the images to 64X64 grid and find the signature of the marker as discussed above. If the signature corresponding to the marker is present in our dictionary of signatures(created in step-1) then it is a valid marker, else not.
Step-8: To determine the pose, I used the `cv2.solvePnP()` to obtain the pose from 3D-2D point correspondence. The 3D points were the world coordinates of the corners of the marker and the 2D points were the coordinates of the corners of the marker relative to the marker itself.

VI. Results and Observations

TASK 2.2:

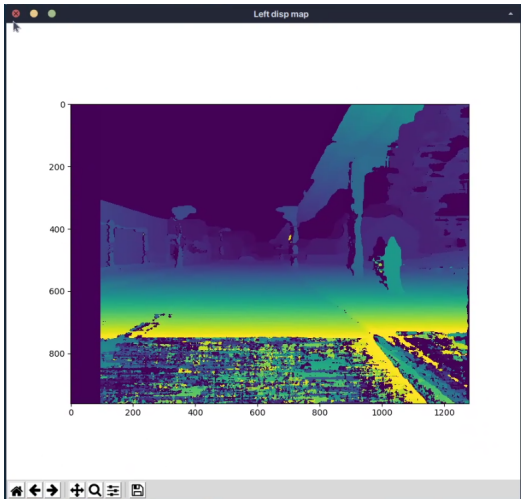

Fig. 9. Disparity map from left image

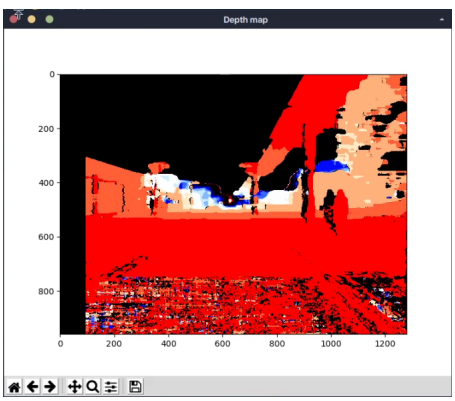

Fig. 10. Depth map

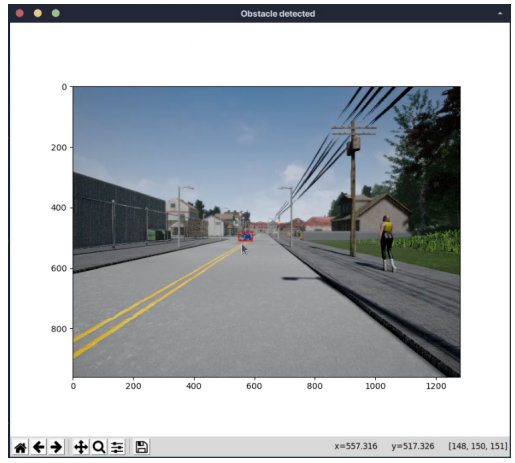

Fig. 11. Obstacle(bike) detection

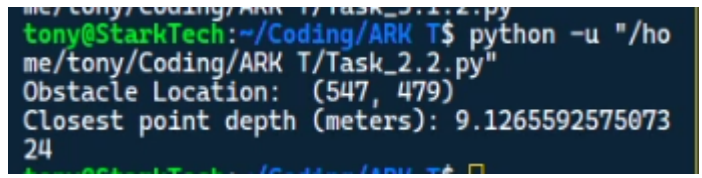

Fig. 12. Final distance of obstacle from camera

TASK 3.1:

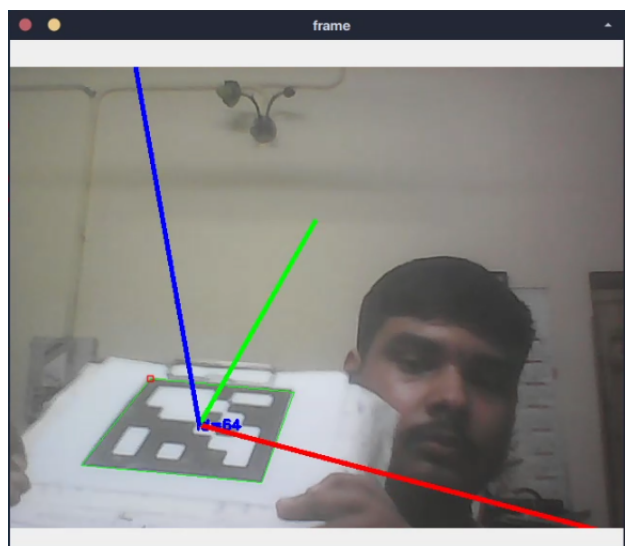

Fig. 13. ArUco detection using inbuilt methods

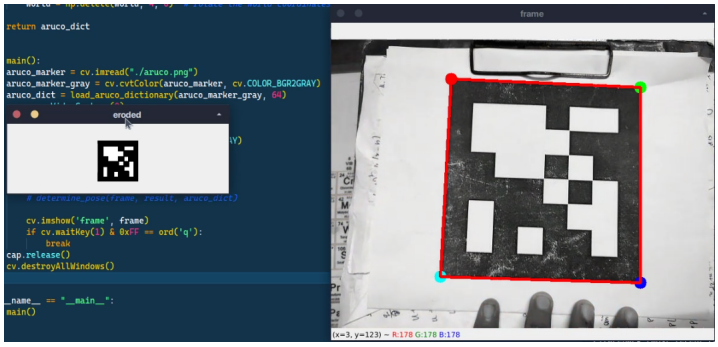

Fig. 14. Square detection from contours

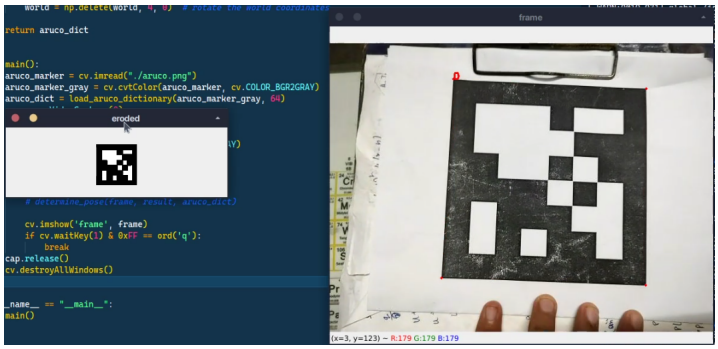

Fig. 15. Upright ArUco detection(notice the 0 on top left corner of the image)

CONCLUSION

I would like to conclude this documentation on a thank you note to the entire team of ARK for curating such an enjoyable yet tricky tasks. Although I could not finish all the tasks within the deadline, I look forward to attempting the remaining problems too in the future.

REFERENCES

[1] https://learnopencv.com/depth-perception-using-stereo-camera-python-c/?utm_source=pocket_mylist

[2] https://aliyasineser.medium.com/aruco-marker-tracking-with-opencv-8cb844c26628

[3] https://youtube.com/playlist?list=PLLSegLrePWgIbIrA4iehUQ-impvIXdd9Q

[4] http://wiki.ros.org/ROS/Tutorials

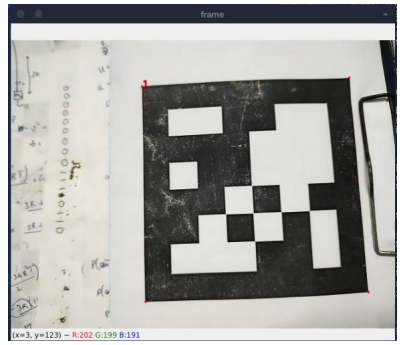

Fig. 16. ArUco rotated 90°clockwise(notice the 1 on top left)

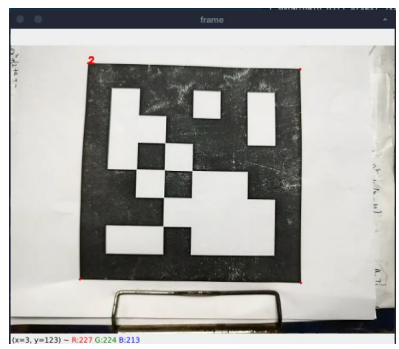

Fig. 17. ArUco rotated 180°(notice the 2 on top left)

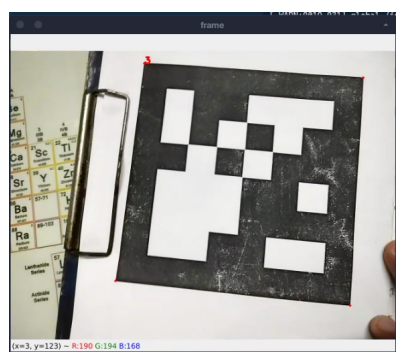

Fig. 18. ArUco rotated 90°anti-clockwise(notice the 3 on top left)