ECE 566 Final Presentation Report
Arjun Viswanathan, John Dale
12/9/23

ArUco Marker Detection and Pose Estimation

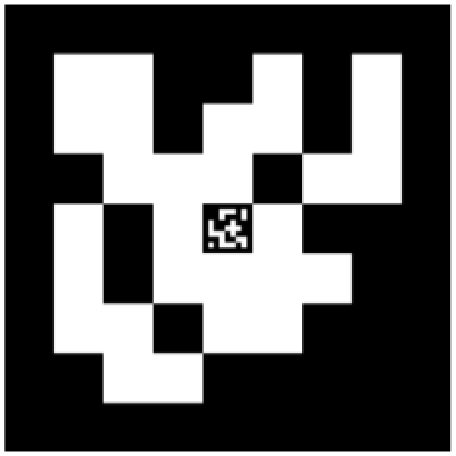
Introduction:

Fiducial markers are very common in Computer Vision and Image Processing. They are artificial landmarks placed in an environment that a camera can easily recognize using trained models and segmentation algorithms. They also encode information within them, which the trained model will extract. ArUco is a specific type of Fiducial marker, which encodes binary numbers. It comes in different shapes and sizes, and so different dictionaries are used with the recognition model to detect them [4]. Sizes include 4x4, 5x5, 6x6, etc, each containing a different number of markers (0-250, 0-500, 0-1000). Dictionaries can also be combined together so we have a highly configurable system [4]. The more landmarks that need to be marked, the more markers are used. So an example dictionary will take the form

DICT_4x4_250

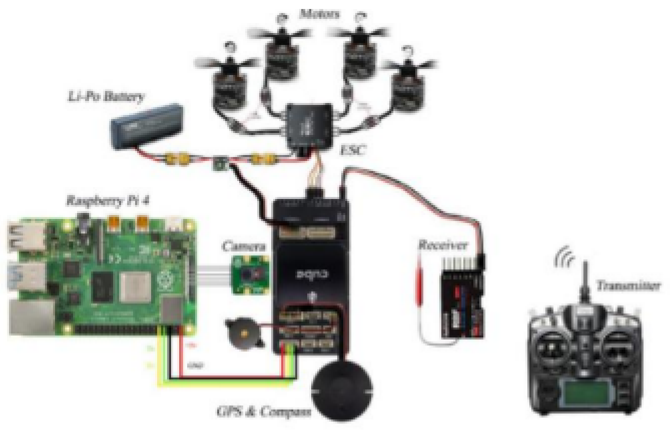
DICT_5x5_1000

An example ArUco marker is shown below.



In this presentation, we consider a research article using ArUco markers for autonomous drone landing. In *Implementation of Robot Operating System in Raspberry Pi 4 for Autonomous Landing Quadrotor on ArUco Marker* by *Atcha Daspan, Anukoon Nimsongprasert, Prathan Srichai, and Pijirawuch Wiengchanda*, Robot Operating System (ROS) is used to create the quadrotor control and the marker recognition. ROS is a publisher-subscriber architecture that acts as a middleware for robot commands through Python in a Linux environment [2]. It is a fully functional platform for developing robots, consisting of programs, libraries, and protocols designed to make building sophisticated and reliable robot systems easier [2].

The literature aims to build, manufacture, and program a quadrotor capable of landing autonomously on an ArUco marker. The system configuration they use in the literature is shown below [2].

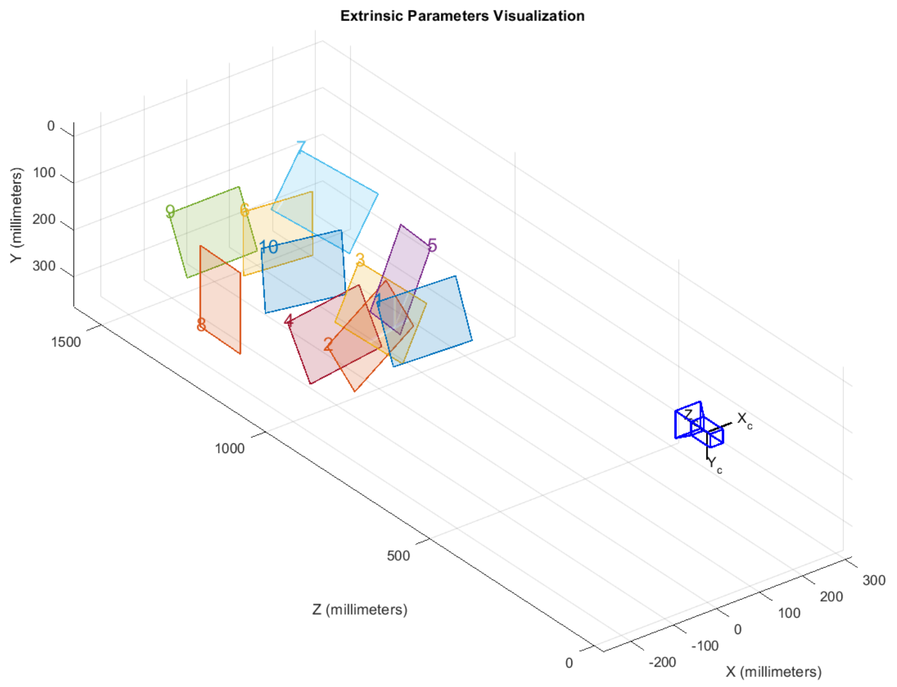


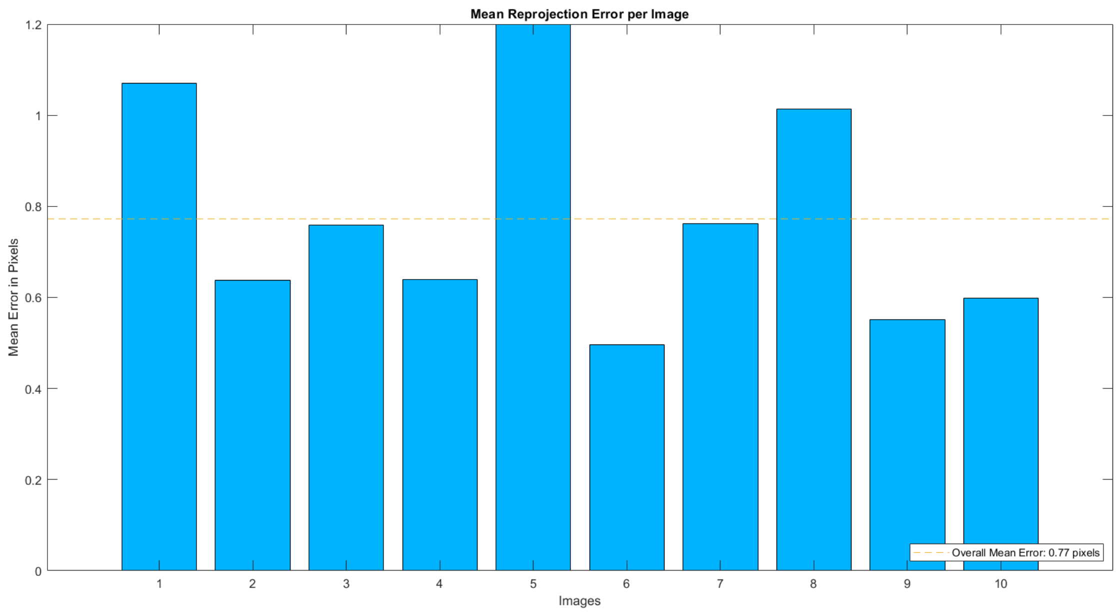
For our presentation, we do not explore the ROS aspect, as it is out of scope of this class. We only do a deep dive into the ArUco marker detection and image processing techniques involved within that aspect. To perform the marker detection and estimate the pose, a number of steps need to be taken. They are outlined as follows:

- MATLAB Camera Calibration
- ArUco Marker Detection
- ArUco Marker Pose Estimation

MATLAB Camera Calibration:

The camera calibration is done so that the camera we use can account for image distortions. Images taken are usually distorted, and as a result the pose estimation can go wrong with an uncalibrated camera [1]. To perform calibration, a number of images are taken. Then, edges are detected and reprojected after accounting for distortion. Then, a matrix of coefficients for the pose estimation as well as distortion coefficients are computed [1]. As for the image, to make it easier for the edges to be detected, a checkerboard is used, because the edges are calculated as where the black and white pixels neighbor each other, as per the gradient method we explored in class. The calibration outputs from MATLAB are shown below. We used 10 images taken of the checkerboard at different angles and distances from the camera. The first image shows how the images are reprojected relative to the camera, and the second image shows the average pixel error in reprojection [1].

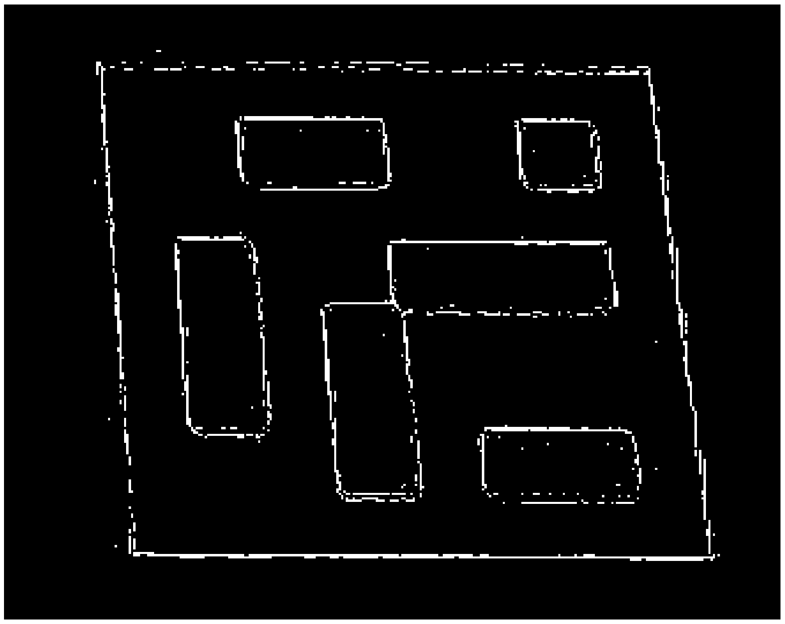




We can see that the reprojection errors are very low, and so the calibration is optimal for detection. The camera matrix and distortion coefficients are sent to the Python code for detection and pose estimation.

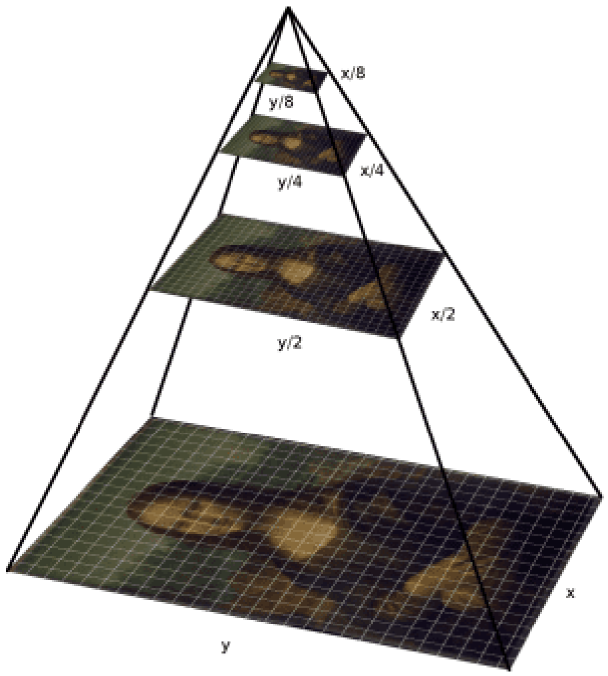
ArUco Marker Detection:

For our marker detection implementation we utilized the OpenCV library and methods. ArUco marker detection in OpenCV begins with initializing and validating parameters. These parameters include the specified ArUco marker dictionary, minimum and maximum values for features, size of the refinement window, and settings for adaptive thresholding [4]. Adaptive thresholding allows for binary conversion based on local pixel intensities, which enhances the future algorithms accuracy for distinguishing markers from the background under varying lighting conditions [4].



Example of adaptive image thresholding output

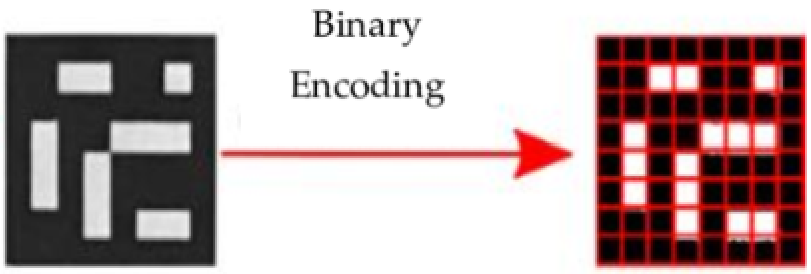
Next, the detection algorithm must build an image pyramid. This pyramid contains a series of images at different resolutions resulting from scaling the input image at many different output levels. The number of levels in the pyramid is determined based on the input image size and the minimum area of a marker that needs to be detected in the application [4]. Therefore, the smallest marker size and maximum distance for which markers are detectable is taken into account [4].



Example four layer image pyramid

The detection process is iterative and finds many potential candidates, which are not necessarily our final detected markers. Rather, the process involves identifying potential candidates through thresholding, contour detection, and shape analysis [4]. Each candidate is located by searching for polygons, specifically warped squares. The image pyramid allows the detection process to occur at varying pyramid levels, which can detect ArUco markers of different sizes and distances. Finally, perspective transformations are performed on the candidates to remove the image warp and return the marker to a square shape. At this point, the unique binary pattern of the marker can be decoded and compared to all markers in our predefined dictionary. Candidates that are too close in proximity to each other or do not match any predefined markers from the dictionary are discarded, and this leaves us with our final detected ArUco markers [4]. In OpenCV's implementation marker corner refinement is added to refine corner positions to sub-pixel accuracy. This process is done by analyzing the neighborhood of pixels near each corner and enhancing localization down to sub-pixels [4].

ECE 566 Final Presentation Report
Arjun Viswanathan, John Dale
12/9/23



Example of ArUco marker binary encoding

While this ArUco marker detection method is robust, there are some trade-offs that must be considered for a specific application. The quality of input image, lighting conditions, occlusions, range of detectable markers, and marker dictionary are all factors that can change the effectiveness and computational speed of the algorithm. Depending on the application, adjustments may be made to provide higher accuracy or further support real-time applications.

ArUco Marker Pose Estimation:

Pose estimation is a computer vision technique with the goal of estimating the correspondence between points with six degrees of freedom in an environment, and their 2D image projections. In our case, we can utilize pose estimation to ensure our drone is descending at the correct trajectory. For our implementation, we utilize OpenCV's pose estimation methods [4].

Pose estimation takes several inputs, including the corners of the detected markers, the length of the marker, the camera matrix, and distortion coefficients. The inputs enable the function to calculate the pose of each marker in 3D space, accounting for camera discrepancies.

The first step involves setting up a 3D reference for a marker in its own coordinate system. This step creates a starting point for our 3D reference, and will preface the true challenge of pose estimation, the perspective-n-point (PnP) problem [4]. PnP involves finding the position and orientation of a camera in 3D space relative to a known object. This is achieved by matching several 3D points, in this case from the ArUco marker, and determining their corresponding 2D points in the camera's image. OpenCV's SolvePnP function is responsible for these operations and returns two vectors, the translational vector (Tvec) and rotational vector (Rvec) [4]. Tvec holds the position of the object in 3D space, which is calculated by a translation of the object's origin relative to the camera's origin. Rvec holds the orientation of the object in space in axis-angle form [4].

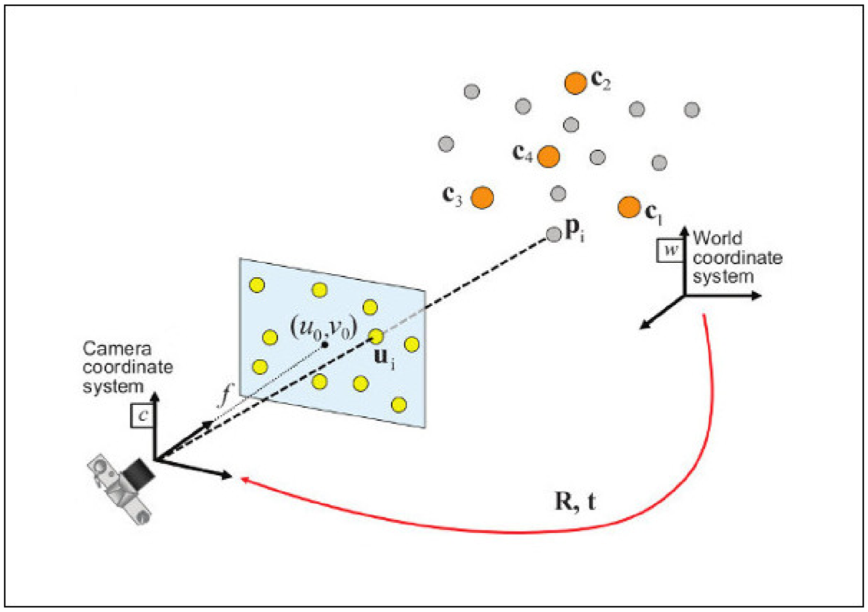
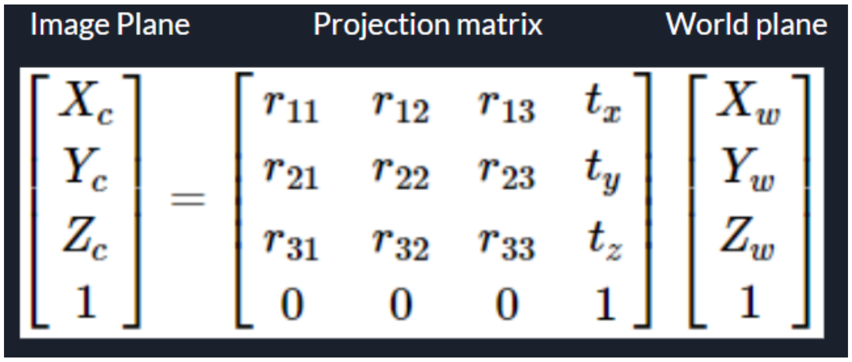


Diagram visualizing the Point-n-Perspective problem

Generating these pose estimation vectors is usually an iterative process which focuses on minimizing reprojection error. Reprojection error is a result of the algorithm guessing position and rotation in 3D space. The error can be calculated by taking the sum of squared differences between the actual 2D marker corners, and the reprojected 2D corners based on the guessed 3D position and rotation [4]. A minimization function can be used to reduce the error below a desired threshold and improve the pose estimation accuracy [4].



$$\begin{array}{ccccc} \text{Image Plane} & & \text{Projection matrix} & & \text{World plane} \\ \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} & = & \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} & & \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \end{array}$$

Matrix required for reprojecting points in the world place (3D) to the image plane (2D)

Pose estimation requires a delicate balance of accuracy and computational complexity for the specified application. To achieve high accuracy, precise camera calibration parameters and accurate marker detection is crucial.

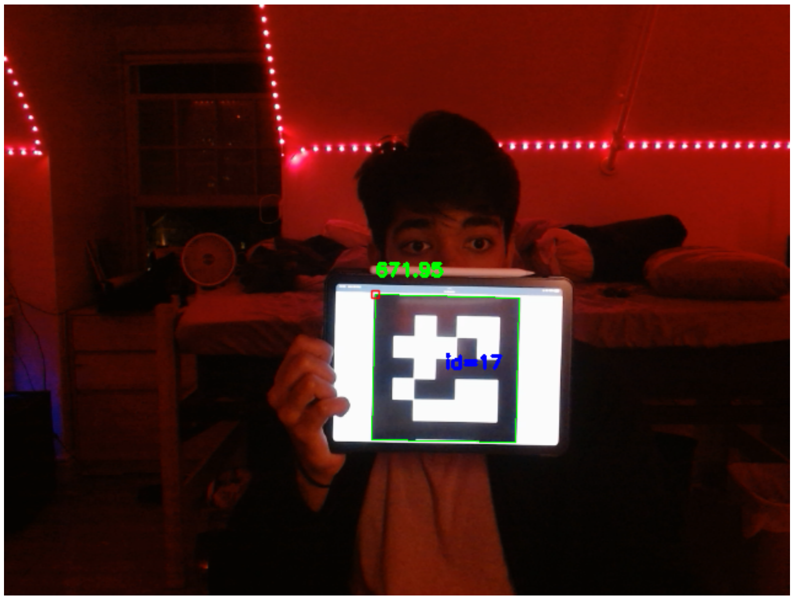
Results:

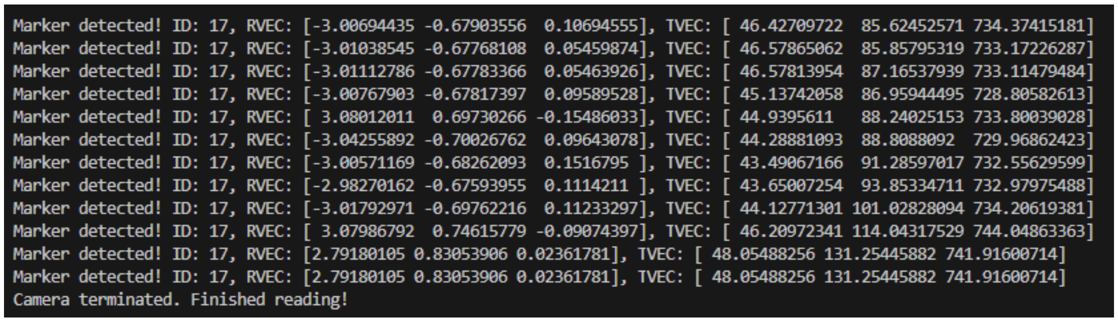
Our ArUco marker pose estimation implementation demonstrates the effectiveness and precision of marker detection and pose estimation, which can be applied to autonomous drone landing. We validated our approach through calibration and testing, with the primary focus being on accuracy and reliability.

1. **Calibration accuracy:** The camera calibration process with MATLAB proved to be highly effective. Calibration reprojection error was averaged at sub-pixel levels, 0.77 pixel reprojection error. This level of error indicates that the intrinsic camera parameters were accurately determined. Low reprojection error in our implementation was crucial in ensuring accuracy for subsequent detection and pose estimation steps.
2. **ArUco marker detection:** Utilizing the OpenCV library, ArUco marker detection was able to perform under various conditions. Adaptive thresholding and image pyramid techniques allowed for markers at different scales and lighting conditions to be detected. ArUco marker detection showed a high success rate in identifying and decoding markers correctly.
3. **ArUco marker pose estimation:** Estimating position and rotation of ArUco markers in our implementation was efficient and accurate. In applications like drone landing, real-time performance is critical to the performance, and in our implementation with a laptop webcam we were able to reach upwards of **30 FPS**. This number can be increased even more with better cameras which will also provide higher resolutions, detecting markers further away. Pose estimation performance heavily relies on the camera parameters and marker detection results, and through our methods described in this paper,

we were able to achieve very high accuracy.

In conclusion, the successful pose estimation of ArUco markers in our project demonstrated the applicability of fiducial markers in assisting to facilitate precise and reliable autonomous drone landings. The combination of MATLAB for camera calibration and OpenCV for marker detection and pose estimation proved to be an effective and fast way to achieve reliable functionality. The results hold promising implications for future applications in autonomous robotic systems and unmanned aerial vehicles. We provide the link to our GitHub repository for this project, which contains all the code necessary to reproduce the marker detection and pose estimation, in the references section below [5, 3].





```
Marker detected! ID: 17, RVEC: [-3.00694435 -0.67903556  0.10694555], TVEC: [ 46.42709722  85.62452571 734.37415181]
Marker detected! ID: 17, RVEC: [-3.01038545 -0.67768108  0.05459874], TVEC: [ 46.57865062  85.85795319 733.17226287]
Marker detected! ID: 17, RVEC: [-3.01112786 -0.67783366  0.05463926], TVEC: [ 46.57813954  87.16537939 733.11479484]
Marker detected! ID: 17, RVEC: [-3.00767903 -0.67817397  0.09589528], TVEC: [ 45.13742058  86.95944495 728.80582613]
Marker detected! ID: 17, RVEC: [ 3.08012011  0.69730266 -0.15486033], TVEC: [ 44.9395611   88.24025153 733.80039028]
Marker detected! ID: 17, RVEC: [-3.04255892 -0.70026762  0.09643078], TVEC: [ 44.28881093  88.8088092  729.96862423]
Marker detected! ID: 17, RVEC: [-3.00571169 -0.68262093  0.1516795 ], TVEC: [ 43.49067166  91.28597017 732.55629599]
Marker detected! ID: 17, RVEC: [-2.98270162 -0.67593955  0.1114211 ], TVEC: [ 43.65007254  93.85334711 732.97975488]
Marker detected! ID: 17, RVEC: [-3.01792971 -0.69762216  0.11233297], TVEC: [ 44.12771301 101.02828094 734.20619381]
Marker detected! ID: 17, RVEC: [ 3.07986792  0.74615779 -0.09074397], TVEC: [ 46.20972341 114.04317529 744.04863363]
Marker detected! ID: 17, RVEC: [2.79180105 0.83053906 0.02361781], TVEC: [ 48.05488256 131.25445882 741.91600714]
Marker detected! ID: 17, RVEC: [2.79180105 0.83053906 0.02361781], TVEC: [ 48.05488256 131.25445882 741.91600714]
Camera terminated. Finished reading!
```

The pictures above show the live detection image frame from our code. This shows the marker highlighted in green, as well as the center and ID in blue, and the distance in mm in green. The second image shows the console output of the ID, Tvec, and Rvec.

ECE 566 Final Presentation Report
Arjun Viswanathan, John Dale
12/9/23

References:

[1] "Camera Calibration - MATLAB & Simulink." *MathWorks*, https://www.mathworks.com/help/vision/camera-calibration.html?s_tid=mwa_osa_a Accessed 9 December 2023.

[2] Daspan, Atcha, et al. "Implementation of Robot Operating System on a Raspberry Pi 4 for Autonomous Landing Quadrotor on ArUco Marker." *International Journal of Mechanical Engineering and Robotics Research*, vol. 12, no. 4, July 2023. *IJMERR*, https://www.ijmerr.com/2023/IJMERR-V12N4-210.pdf Accessed 9 December 2023.

[3] Eser, Ali Yasin. "ArUco Marker Tracking with OpenCV | by Ali Yasin Eser | Medium." *Ali Yasin Eser*, 27 June 2020, https://aliyasineser.medium.com/aruco-marker-tracking-with-opencv-8cb844c26628 Accessed 9 December 2023.

[4] "OpenCV: Detection of ArUco Markers." *OpenCV Documentation*, https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html Accessed 9 December 2023.

[5] ECE 566 Final Project, GitHub repository arjuns-code-center/ArUco_Detection_and_Pose_Estimation (github.com)