Northeastern University

EECE5554, Robot Sensing and Navigation

12/11/2024

ArUco Marker Tracking Robot

Group Number: 4
Arjun Viswanathan, Patricia Meza, Maria Apostle, Rahul Sha Pathepur Shankar

1 Problem Addressed

Virtual autonomous solutions for learning and communication are still developing. When COVID took over the world, a lot of schools, families, and companies struggled to adapt to the fully online way of life. Resources were missing to provide a stable and engaging approach for young children to learn, or for people to tour virtually. We propose an autonomous target-tracking robot capable of navigating static obstacles in a representative environment while tracking an ArUco marker. This robot fills in for a human holding a camera, and is meant to be used to fill the gaps that the pandemic exposed. This project is important because it paves the way to more interactive online solutions for humans to be present even when they are not physically present, and putting robots with humans in a cooperative environment. This robot can be used in video lectures, virtual tours, and poster sessions.

2 Background

2.1 Hardware Overview

In this project, we utilize Turtlebot3 Burger. It provides a compact and modular platform, making it ideal for implementing and testing autonomous navigation and target tracking algorithms. The Turtlebot3 is equipped with Raspberry Pi 4B, RPLidar A1, OpneCR 1.0, LiPo Battery, Logitech C920 Web Camera, and Dynamixel XL430-W250 servo motors provided to us by the EECE department.

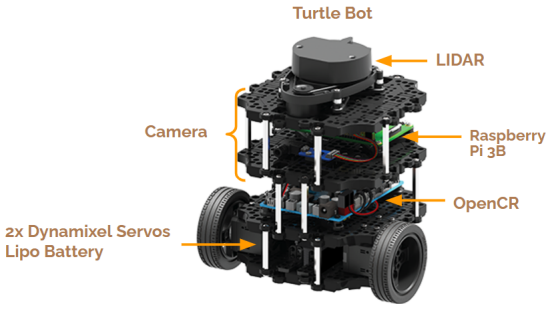

Figure 1: Turtlebot3 Burger

2.2 OpenCV and ArUco Markers

The main software for computer vision is OpenCV, an open source computer vision and machine learning software library. The type of targets we are tracking are called ArUco markers. An ArUco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id) [2].

For the purpose of our task, we will only be using one marker with ID 1 as the robot is tasked to keep track of one target. For detecting the ArUco markers, we used the ArUco package from the cv2 library. The camera detects the ArUco marker to aquire pixel coordinates of the detected marker corners in the image (e.g., $[x_1, y_1], [x_2, y_2], ...[x_4, y_4]$). We also have information on the physical marker side lengths, and specific camera parameters such as a 3x3 matrix containing focal lengths and optical center and distortion coefficients for the camera lens. Equation (1) represents the **camera projection model**, which maps 3D world coordinates (X, Y, Z) into 2D image coordinates (x, y) using the camera's intrinsic matrix $\mathbf{K}$, rotation matrix $\mathbf{R}$, and translation vector $\mathbf{t}$. The CV2 library uses the Perspective-n-Point (PnP) algorithm to minimize the reprojection error. It measures the difference between the observed 2D coordinates (x_i, y_i) and the projected 3D points, which is minimized to estimate the camera pose (rotation $\mathbf{R}$ and translation $\mathbf{t}$).

$$\begin{bmatrix} \hat{x} \\ \hat{y} \\ 1 \end{bmatrix} = \mathbf{K} \cdot \left(\mathbf{R} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \mathbf{t} \right) \tag{1}$$

2.3 Navigation2 Library

We use the Robot Operating System (ROS) 2 Nav2 library to perform our path planning, smoothing, and control. The Nav2 library provides servers for each aspect of navigation, which we elaborate on in the methods section. This outputs the linear and angular velocities for the robot to follow, which is sent over serial to the OpenCR board. We use an additional node on top of this to automate the navigation and target tracking process. This node takes the ArUco marker pose information and convert it to a goal pose.

3 Methods

We use the following software block diagram to implement our work. In this diagram, the target detection works with the navigation to make our autonomous system work as intended.

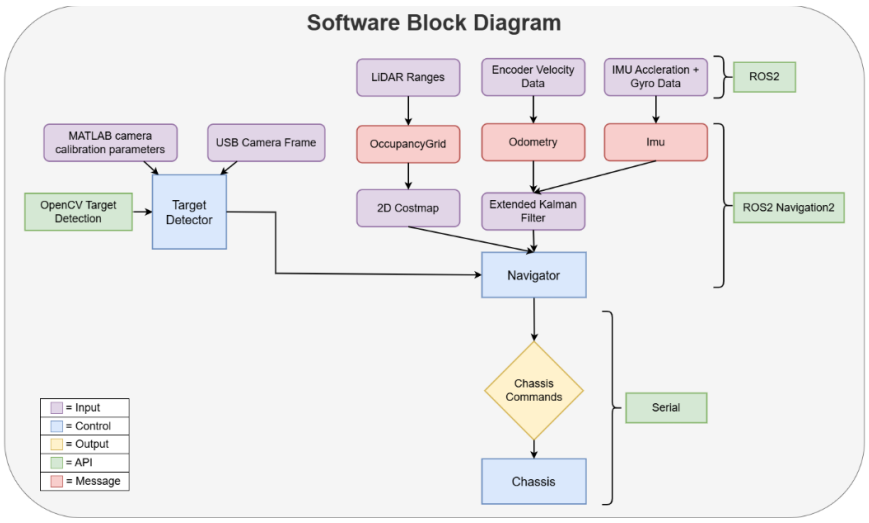

Figure 2: Software Block Diagram

3.1 ArUco Marker Tracking Implementation

Camera calibration was performed using Matlab's Computer Vision Toolbox which includes various tools for calibration with a known calibration pattern. In our case we used a printed out checker board and took 20 images with the USB camera. The Matlab toolbox includes functions to detect the pattern, generate world coordinates for the planar pattern keypoints, and calibrate the camera. Figure 3 shows the output of the MATLAB script which includes a visualization of the extrinsic parameters, and produced 2 .mat files containing the camera matrix and distortion coefficients that will be called by the ROS node.

The ROS node named `ArucoPublisher`, which handles the computer vision portion of the robot, follows the following structure. The publisher is created, and the camera calibration parameters are declared from a `.mat` file. An ArUco detector is initialized for a specific library of ArUco IDs (`cv2.aruco.DICT_5X5_50`). The camera object is initialized, and the USB camera is activated. Frames are continuously read in a loop at 10 Hz using `camera.read()`, which outputs the frame itself and a Boolean operator indicating whether

a frame is successfully returned. If a frame is returned, the node calls `cv2.aruco.detectMarkers` to check for markers. This function outputs the corners and IDs of all the markers detected in the frame. If a marker with `ID == 1` is detected, the pose is calculated using the camera coefficients, marker size, and the location of the corners in that frame. The publisher then publishes this data as a vector containing the `[x,y,z]` translation coordinates.

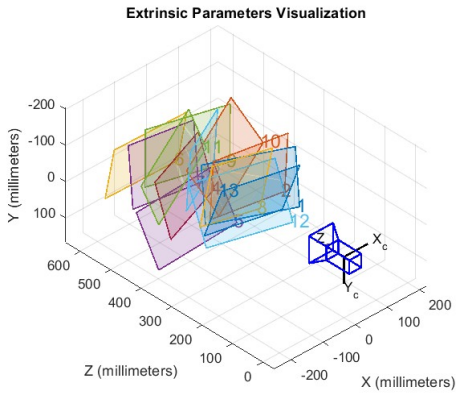

Figure 3: Matlab figure produced by computer vision toolbox

3.2 ROS2 Coordinate Transforms

Coordinate Transforms are what allow ROS to interpret the robot's movement. Each component contains its own transform relative to the base and is connected using joints in the Unified Robot Description File (URDF), which contains the physical description of the robot. On top of the URDF transforms, we also provide the odom to base transform using the fused sensor data from the Extended Kalman Filter (EKF), and the map to odom transform using Simultaneous Localization And Mapping (SLAM).

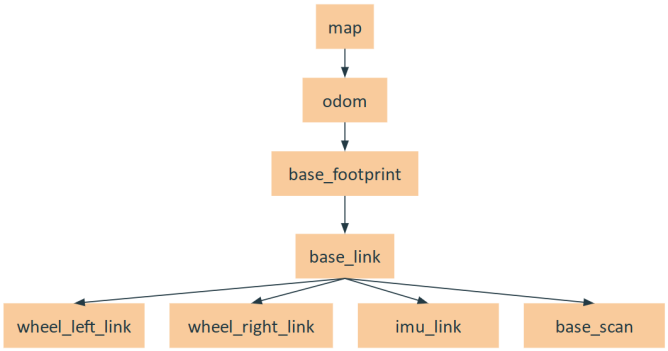

Figure 4: Coordinate transform flowchart

3.3 Nav2 Behavior Tree

In Nav2, behavior trees determine the internal logic and decision-making for the navigation system. For this system, the navigate with re-planning and recovery behavior tree was used. This runs 2 nodes: navigation and recovery. Navigation computes a path, smooths it, and then runs the follow path command. If that fails,

it will go to the recovery behaviors in the recovery node. These recovery actions involve resetting costmaps, turning, waiting, and backing up.

3.4 Nav2 Servers

The Nav2 servers are used to coordinate the overall navigation process. For this project, the behavior, smoother, planner, controller, and BT (Behavior Tree) navigator servers were used, with their parameters modified to suit our robot.

Behavior Server: The behavior server sends commands to run the recovery behaviors within the behavior tree. It uses the local costmap to trigger the recovery actions when the robot is stuck while navigating (for example, if obstacles are too close).

Smoother Server: The smoother server takes the path computed by the BT Navigator and smooths it out to avoid jagged movement. For the purposes of our project, the Constrained Smoother was used. This generates an intelligent smoothed path that avoids obstacles along the way to some radius. It also uses the global costmap to compute the best path with the lowest costs.

Planner Server: The planner is used to plan the best route. This project uses A* search, a widely-used search algorithm.

Controller Server: The controller takes the path computed from the BT navigator and provides wheel velocities for the robot to move along the path. We use the Dynamic Window Approach planner (DWB Planner). This is a critic-based planner which allows the user to define any number of critics (base obstacle, goal align, rotate to goal, path align, etc.), allowing it to be configured for any robot type.

BT Navigator: Lastly, the BT Navigator ties the navigation servers together. It uses both the behavior tree and Nav2 plugins for recoveries, movement, and speed to compute the path to pose actions as well as follow the path actions. It also receives the fused odometry and the goal pose set.

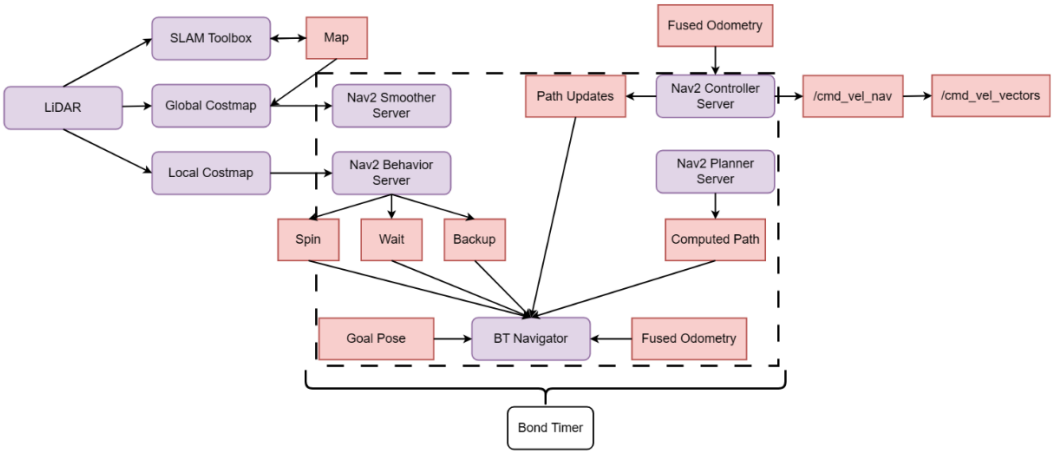

Figure 5: Navigation flowchart

3.5 ArUco Marker Pose to Goal Conversion

Once the ArUco marker pose is established, it is passed over to the `nav2pose` node in order to determine a goal. `nav2pose` calculates the goal pose based on the velocity of the target in order to ensure smooth tracking. If the goal pose were in front of or at the target position, the target would have moved further away by the time the robot reached the goal, leading to a jagged start-stop motion.

Therefore, the goal must be set some distance behind the target, depending on how fast it is moving. The velocity of the target is simply calculated by dividing the change in forward position by time, and the

resulting robot velocity is calculated as follows:

If target velocity is less than 0 (coming towards robot):

$$v_{robot} = -1.5v_{target}^2 \tag{2}$$

Else (moving away from robot):

$$v_{robot} = 2.5v_{target}^2 \tag{3}$$

3.6 Turtlebot Serial Communication

Our Robot employs serial communication between Rasberry Pi 4B, and the OpenCR 1.0 board. The Rasberry Pi sends velocity commands (linear and angular velocities) over the serial communication to the OpenCR 1.0 board, and receives encoder and IMU data back for odometry computation.

4 Results

4.1 Target Tracking Accuracy

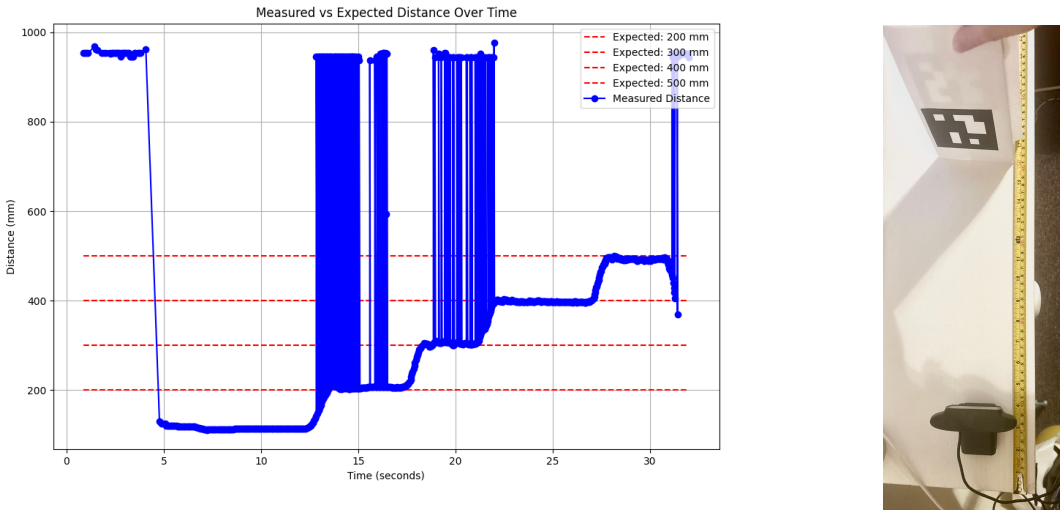

Figure 6: Accuracy test setup using a USB camera, tape measure, and an ArUco marker

A test was performed to evaluate how accurately the pose of the marker could be estimated using the camera. The marker was held at distances of 10, 20, 30, 40, and 50 cm for a short period. The results, displayed in Figure 6, show that the position estimation was mostly accurate. However, there were numerous outliers indicating a distance of around 900 meters. It was found that the camera was falsely detecting an ArUco marker with a different ID in the crowded background of the image. This highlights the importance of specifying the marker ID in the code to avoid mistakenly identifying noise or unintended markers as valid detections.

4.2 Navigation with SLAM

On the physical system, we noticed that the Raspberry Pi 4 was too weak to perform the navigation and computer vision at a good rate. Due to the Pi 4 system limitation, we noticed that the robot was not able to plan and execute smooth paths as it was constantly lagging due to lack of system resources. As a result, of this, we decided to switch to a simulated work and use a more powerful computer so we could test out the navigation and our computer vision nodes. We show a SLAM generated map of a physical test environment and a simulated navigation run's start and end of the robot's navigation path.

For the future, we can implement the computer vision tracking at a lower rate and use velocity of target tracking just like velocity of target navigation, which will allow us to track with lower frame rates and achieve good system performance in the constrained system we have.

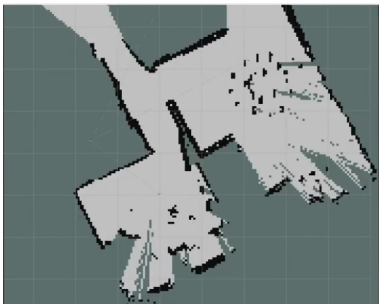

Figure 7: SLAM generated map of a room and living room in the physical world

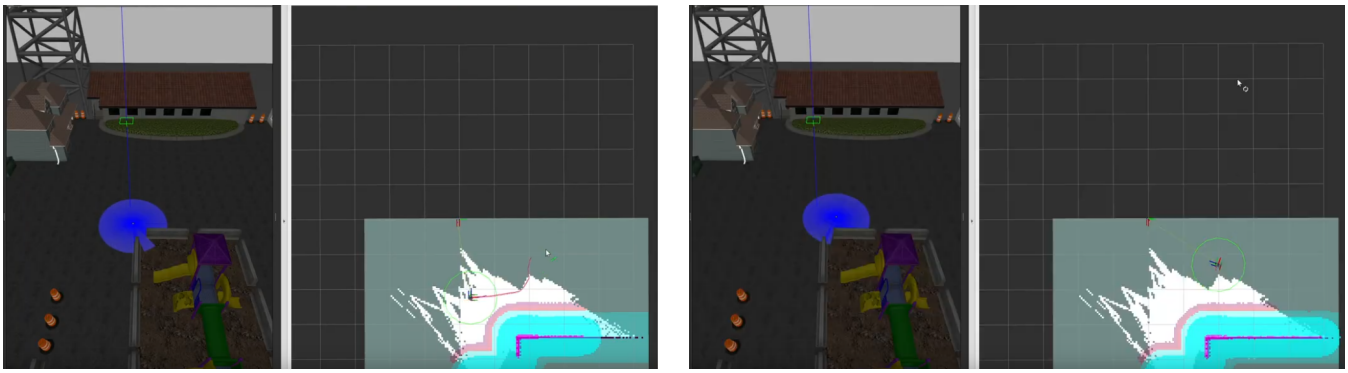

Figure 8: Simulated Navigation using SLAM

5 Conclusion

In this study, we learned how to use sensors like the IMU, LiDAR, and wheel encoders to perform sensor fusion using the EKF. We also learnt to use the ROS2 Nav2 libraries to perform autonomous navigation around static obstacles in a representative environment. We performed SLAM using LiDAR to keep updating our map as we move around and give way to dynamic obstacle navigation. This study allowed us to think critically about possible issues that may come up in the field of robotic navigation and computer vision, as well as tools to debug them.

References

[1] Atcha Daspan et al. “Implementation of Robot Operating System in Raspberry Pi 4 for Autonomous Landing Quadrotor on ArUco Marker”. In: *International Journal of Mechanical Engineering and Robotics Research* 12.4 (2023).

[2] *Detection of ArUco Markers*. OpenCV Documentation. 2024. URL: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html.

[3] Yuxiang Liu et al. “Autonomous Vehicle Based on ROS2 for Indoor Package Delivery”. In: *2023 28th International Conference on Automation and Computing (ICAC)*. 2023, pp. 1–7. DOI: 10.1109/ICAC57885.2023.10275227.

[4] *Navigation 2 Library Documentation*. URL: https://docs.nav2.org/.

[5] Adrian Rosebrock. *Detecting ArUco markers with OpenCV and Python*. Dec. 2020. URL: https://pyimagesearch.com/2020/12/21/detecting-aruco-markers-with-opencv-and-python/.