April Bot

Em Kramer, Alex Wan, Adam Yaj

Introduction

Robot perception is a core robotics theme that enables decision-making processes and human-robot interaction. One particular type of perception that further encourages human-robot interaction is visual perception. The ability to relay visual cues to a robot is a convenient way of communicating and controlling a robot's actions in the field, providing a wide range of applications in mobile robotics. A specific area of robotics where this may be useful is utilizing robots for human services, as it allows for interaction with a robot using visual cues, giving the robot the ability to complete tasks indicated by the human. Additionally, the ability to navigate to identified structures in an unknown environment has great value for real-world applications.

This project addresses these areas of interest by implementing a gesture-controlled robot capable of navigating through a simulated environment using AprilTags and Simultaneous Localization and Mapping (SLAM).

Relevance to CSCI 4551

The robot uses robot vision, onboard sensors (LiDAR, Odometry Sensors), and a SLAM algorithm to map and navigate its environment, with a ROS2 framework. Additionally, computer vision with feature detection was implemented to distinguish different hand gestures indicated by the user through a webcam. All of these topics are discussed within CSCI 4551 and are implemented in practice through this project.

Methodology

The ROS system can be broken down into six main parts: Gesture Control, AprilTag Detection, SLAM Navigation, Goal Detection, the Gazebo World, and System Integration. For a brief high-level overview of the system, the robot receives sign language gestures from the user's webcam, specifically the numbers 0 through 10. Each number corresponds to a unique AprilTag ID that has been texture-mapped across objects within the gazebo world. Given a command, the robot uses SLAM and onboard sensors to search for the correct AprilTag until it is found while also avoiding collisions with all obstacles within the world.

Gazebo World (Alex)

The gazebo world was created based off of the empty world file. Simple cubes and walls were created, and AprilTag images were texture-mapped across one face of the cube using Blender. The cubes were imported into the world and the turtlebot was added to the launch file along with ROS-gazebo bridges.

Gesture Control (Adam)

Hand gestures are identified through the *hand_gestures* ROS2 package. This package contains two nodes, *webcam_publisher* and *gestures*. The webcam publisher node establishes a constant stream of images to the */webcam_raw* topic. The *gestures* node subscribes to this stream and runs mediapipe hand landmark detection on each frame. The returned set of landmark points is used to determine gestures. Our implementation utilizes American Sign Language gestures for numbers 1 to 10. To do this, each number was linked to a list of five Boolean values; true values represented extended fingers. This was stored in an indexed list, making a 2D matrix, for easy number lookup. To detect an extended finger, four points were used: reference, tip, pip, and mcp. The tip was the top of the finger, the pip the last joint before the tip, the mcp the joint connecting the finger to the palm, and the reference was usually the palm base. Pixel distances were measured from the tip, pip, and mcp to the reference point. The difference in distance to the reference point of the tip and the pip determines extension. If the tip is further than the pip, then the finger is extended. A small threshold is applied to this to allow for small errors and imprecise finger movement. The mcp is used to scale this distance. The difference between tip and pip is divided by the distance to mcp to remove variation with distance. The thumb works the same, but with the reference point being the mcp of the pinky and the pip being the mcp of the thumb. A larger threshold is also used to account for incomplete thumb retraction. This gives us an identification value between 0 and 11, where 0 is an unknown gesture, 11 is our stop gesture, and 1 to 10 are our goal IDs. When no hands are detected, a value of -1 is passed. These values are published to the */gestures* topic.

Goal Detection (Alex)

The goal node was created to serve as a buffer between receiving gesture commands and sending controls to the robot. Once a gesture is displayed on the webcam frame for a certain amount of time, a goal ID is published to the */goal_id* topic. This prevents incorrect gestures from accidentally controlling the robot. The goal node also checks if the goal has been reached. If so, the node will publish a stop command to the */goal_id* topic.

AprilTag Detection (Em)

AprilTag detection is performed through the detector node in the april_tag_detector package. This node is subscribed to the */camera/image_raw* topic to receive raw camera frames and the */camera/camera_info* topic to calibration parameters necessary for calculating the AprilTag's position. When a frame is received from */camera/image_raw*, CV Bridge converts the ROS Image message to OpenCV format. The detector then applies adaptive thresholding to create a binary image, then uses hierarchical contour detection to find potential tags, since

AprilTags have child contours as the inner pattern of the tag. Each quadrilateral contour is validated by checking if it has four vertices, a sufficient black border, and child contours. The valid candidates then undergo perspective transformation to create normalized 240x240 pixel square images. The detector then divides the square into an 8x8 grid and samples each cell to extract binary bits, black pixels becoming 1 and white pixels becoming 0. After disregarding the outer border, the inner 6x6 region provides a 36-bit code representing the AprilTag's identity. This code is rotated through four orientations and compared against preloaded templates using Hamming distance, measuring the similarity between detected and template codes. A match that has at most a 5-bit difference is accepted as a valid detection. The detector then performs 3D pose estimation on the identified tags, which takes the known 3D corner positions, detected 2D corner pixels, and camera calibration data to compute the tag's position and orientation relative to the camera. The resulting rotation and translation vectors are then converted from OpenCV's coordinate system to ROS's, and the rotation matrix is converted to a quaternion. The detector then publishes on three topics: individual PoseStamped messages to */apriltag_detections* containing the pose for each AprilTag, an AprilTagDetectionArray to */april_tags* containing all detected tags with their IDs and poses, and an annotated visualization image to */apriltag_detections_image* with green outlines and ID labels on the detected tags.

The video_viewer node provides real-time visualization of what the robot's camera sees by subscribing to */camera/image_raw* for the raw camera feed and */apriltag_detections_image* for the annotated feed with detection overlays. When an Image message arrives, CV Bridge converts the message from ROS format to OpenCV format. The node will then display either the annotated image or the raw image, prioritizing the annotated image if available.

SLAM Navigation (Em)

The navigator node in the april_tag_navigator package runs the autonomous navigation system. This node is subscribed to three topics: the */map* topic for the SLAM-generated occupancy grid, */scan* for LiDAR obstacle detection, and */april_tags* for tag detections. The occupancy grid from */map* is reshaped into a 2D array where -1 represents unknown space, 0-49 represents free space, and 50-100 represents obstacles. The map metadata provides the resolution and origin needed to convert between world coordinates (meters) and map coordinates (grid cells). Robot localization comes through TF2 rather than direct topic subscription. Every 0.2 seconds, the navigator queries the transform from 'map' to 'base_footprint' frame, obtaining the robot's current position and orientation. When AprilTag detections are received, the navigator looks up each tag's transform from the map frame to get its global position rather than camera-relative position. These positions are then adjusted to account for tags mounted on walls by stepping from the tag toward the robot until free space is found, which are then stored in a dictionary to be used for path planning.

The navigator receives goal commands through */goal_id* as Int32 messages representing the target AprilTag ID. When it receives a goal, the navigator transitions to its PLANNING state

and clears any existing path information. If the tag exists in the discovered tags dictionary, direct A* path planning is attempted to a position in front of the tag. If the tag is unknown or unreachable, the navigator begins to explore the environment using frontier-based planning. Frontier detection scans the map for free-space cells adjacent to unknown cells within a 15m radius. These boundary cells are clustered together, then scored based on cluster size, number of unknown neighbors, and distance from the robot. The navigator will then attempt to plan A* paths to the top-scoring frontiers, using the first reachable one. The A* algorithm operates on the occupancy grid, using a priority queue for the open set and maintaining g_scores (cost from start) and f_scores (cost plus heuristic). The Euclidean distance heuristic is used to ensure optimal paths. For frontier exploration, the navigator treats unknown cells as traversable.

Once a path has been found, the navigator transitions into a NAVIGATING state. In this state, the robot follows the waypoints (computed by A* path planning) by computing the distance and angle to the next point, and then publishing the necessary velocity commands to reach that given waypoint. While following the path, if the robot gets too close to an obstacle, where it has a chance of colliding with it, the navigator will transition into its RECOVERY state. In this state, the navigator causes the robot to back away from the obstacle and forces the navigator to generate a new path to a new frontier. Once the robot has reached the end of a path and it doesn't know where the target AprilTag is, the navigator enters the SCANNING state, where the robot will then rotate 360 degrees to ensure full observability before continuing onto the next frontier.

When the goal AprilTag is visible, the navigator enters its TRACKING state. In this state, the navigator plans a path directly to the AprilTag's position. This path will be replanned if the goal moves more than 0.2m. Once the robot gets to the end of the path and is within the approachable distance of the tag, the navigator then transitions to the REACHED state. In this state, the navigator publishes "True" to */reach_goal* to indicate that it has successfully reached the targeted tag.

Throughout the system processes, the navigator maintains a JSON database of the discovered tag positions that is loaded on startup and saved on shutdown. The navigator also visualizes the process in RViz2. The navigator publishes the path, the tag locations, the map, and the frontiers to RViz2 to visualize. The path is red when it is a possible path, while it is green when that is the path the robot is going to follow. The tags are represented by a green square. The frontiers are represented by blue circles when they are considered candidate frontiers and are yellow when the robot is attempting to path to them. All the frontiers are also labeled by their rank, indicating the order in which the navigator will attempt to path to them.

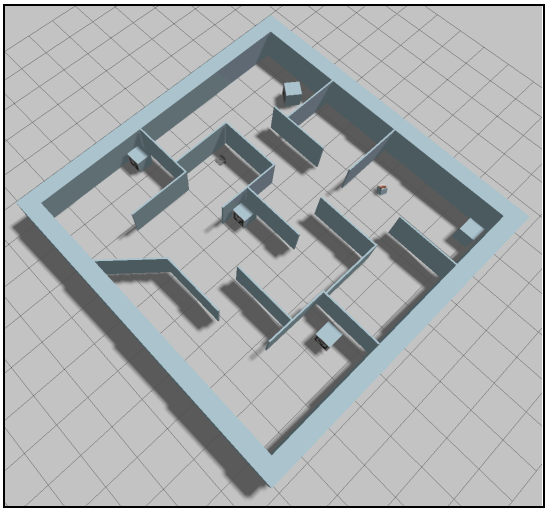
System Integration (Adam)

Our team manages code with Git using GitHub repositories to store the codebase. We had hour-long meetings two to three times a week to discuss project progress. This permitted quick pivoting of ideas when issues arose. Additionally, this allowed for easier system integration due

to great communication. Integration happened at three layers: local, subsystem, and system. Local integration involved each person individually implementing their own system, our Gesture Control, AprilTag Detection, SLAM Navigation, Goal Detection, and the Gazebo World. Each local integration had its own branch in GitHub. Once complete, pull requests and meetings were used to resolve any issues and merge code. Subsystem integration occurred when further development of individual ideas required other local integrations. For instance, AprilTag Detection relies on the Gazebo World for 3D AprilTag and cameras, and Goal Detection relies on Gesture Control for gesture input. This integration was handled by the person creating the reliant packages, such as Goal Detection and April Tag Detection. Full system integration happens at the end and is managed by a simple package, *april_bot_system*, which launches each subsystem. This package made launching the whole system easier. Full system testing occurred in the *dev* branch. At this level, it is just managing simple issues, such as dependencies, topic names, and documentation.

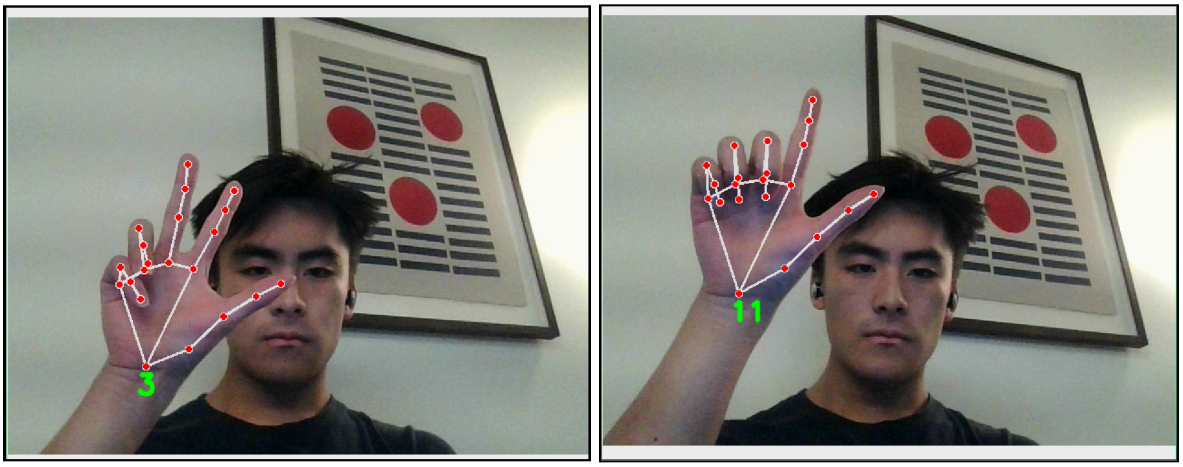
Results

Gazebo World (Alex)

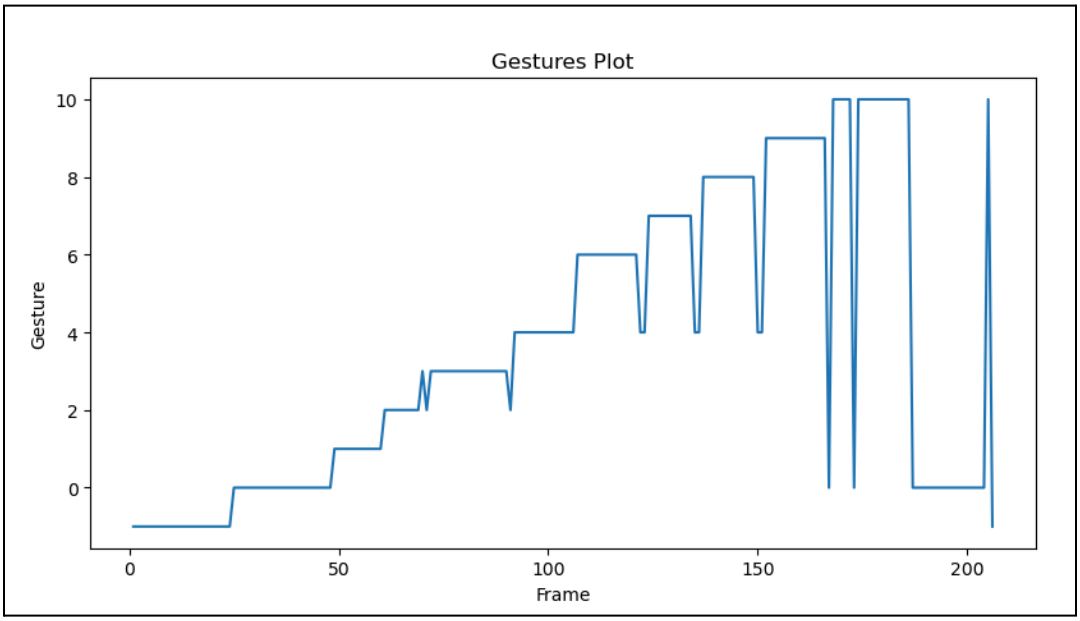


Accomplished: An enclosed static environment was created with five AprilTags attached to objects placed across the map. Turtlebot (specifically Waffle) was added to the world. All wall models and cubes were successfully modeled in Onshape and processed in Blender.
Limitations: The world lacks the depth a real-world environment would have (dynamic lighting, dynamic obstacles, greater size).

Gesture Control (Adam)



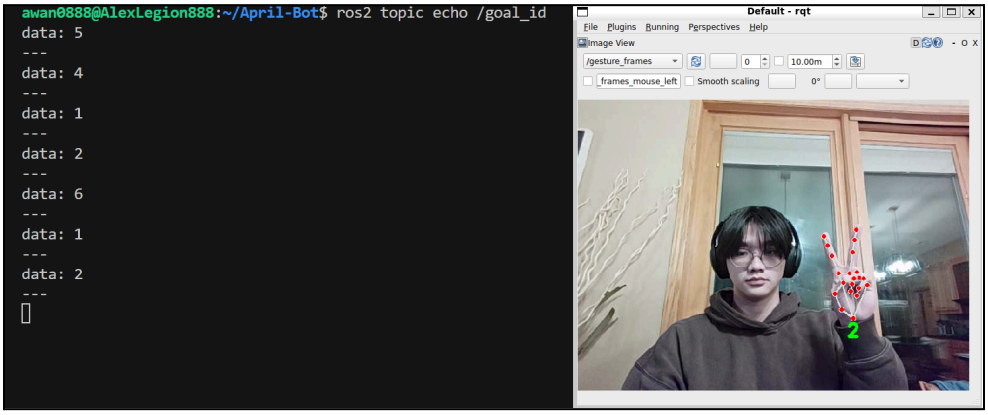
Accomplished: The gesture detection works very well in detecting ASL numbers from 1 to 10. This implementation is fast, allowing us to work with the full frame rate of the webcam. This allows the detection of gestures with a moving hand, assuming the shutter speed is fast enough. The figure below shows the gestures detected when counting to 10 from 1. The finger extension detection works well. As it uses reference points instead of measured gestures, it detects any orientation, along with left or right handedness. Additionally, this extension detection allows easy addition of new gestures. To define a new gesture, add a new array to the existing gestures list, with 1 representing extended and 0 not extended.



Limitations: The finger extension algorithm works only with static gestures. So, gestures, such as ASL above 11, fail detection. This limits our total gestures to 32 maximum. Additionally, likely because of how the models for palm and finger detection are trained, upside-down gestures are less accurate. Hand landmarks are falsely evaluated or not detected.

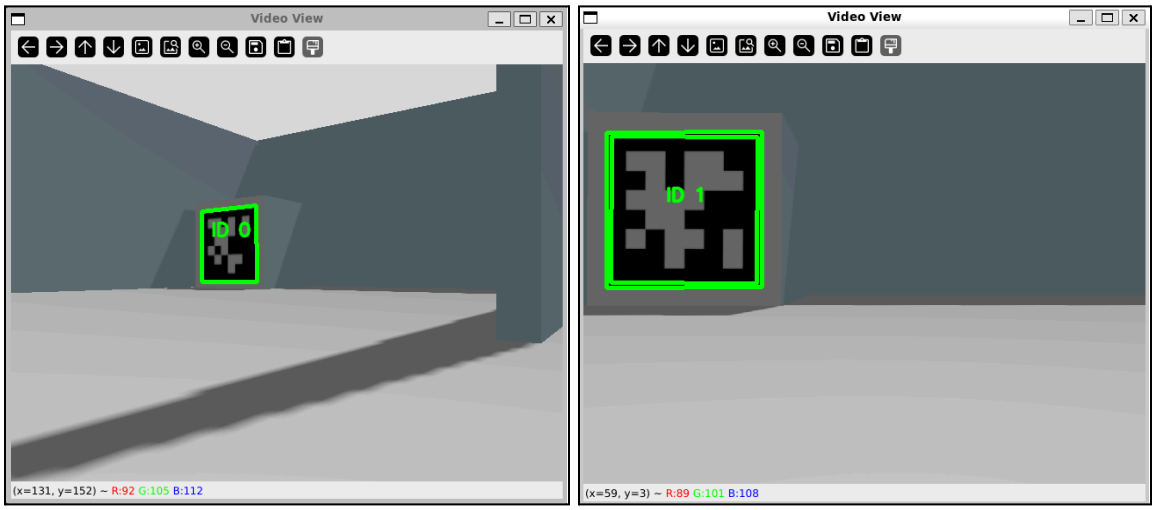
Fail Cases: The hand landmark detection and extension detection fail due to the hand orientation. The hand has to be head-on, palm or backhand facing the camera; otherwise, the point detection fails to detect, or the threshold distance fails to correctly scale. The gesture detection is also limited in distance. When the hand is too far, features are no longer well defined, making it struggle and fail. When the hand is too close, the hand is cut off, and the estimation fails. When the hand is not distinct, such as matching the background or dark lighting, the detection fails. The last failure comes from hand obstruction. The landmark detection will attempt to predict it, but it will fail when too many fingers are obstructed.

Goal Detection (Alex)

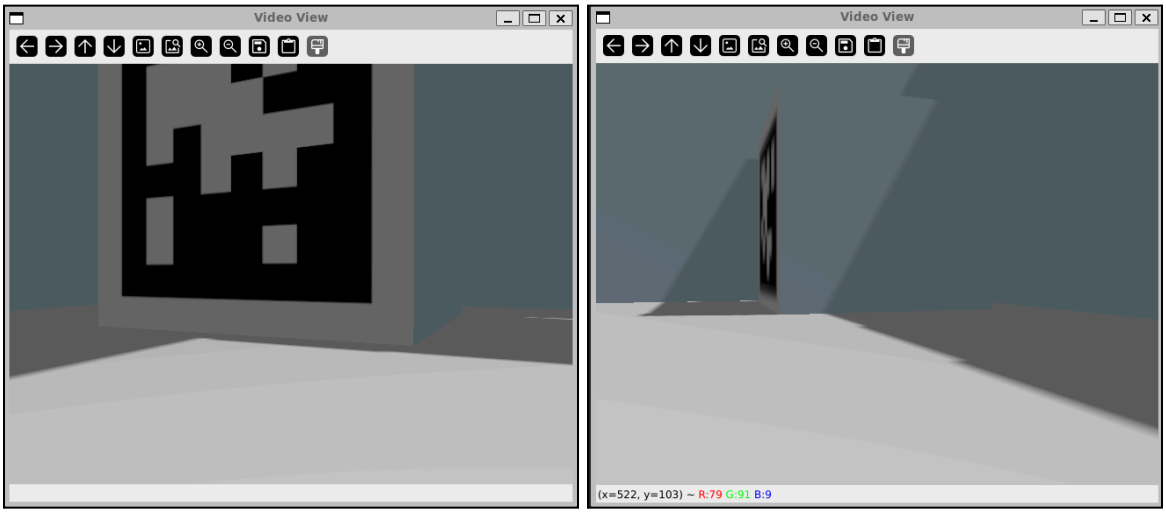


Accomplished: The goal node successfully times how long a valid gesture has been held up for, and if the time is greater than a set time amount, the corresponding goal ID will be published to the */goal_id* topic. Switching gestures will reset the timer.
Limitations: It is difficult to judge how long a gesture has been held up for.

AprilTag Detection (Em)



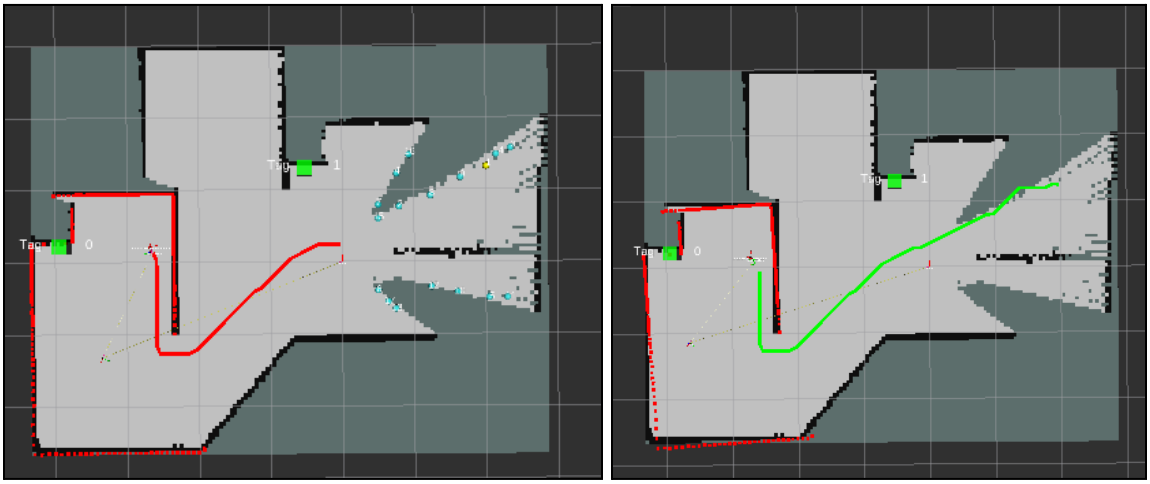
Accomplished: The system is able to accurately detect AprilTags in the robot's environment through the robot's camera.

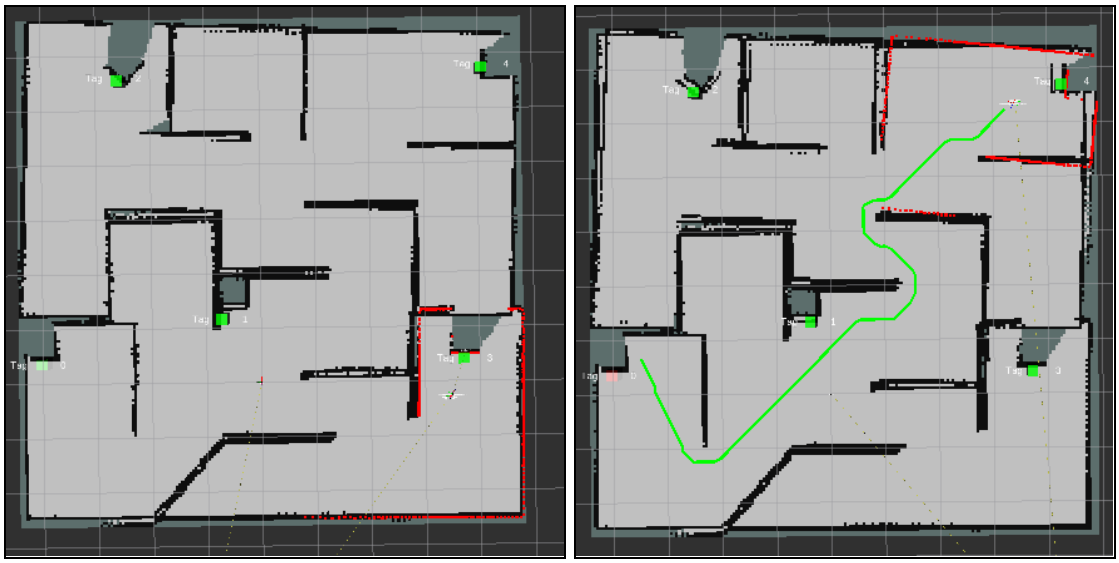


Limitations: The AprilTags must be completely within the camera's frame and the inner tag patterns must be visible and recognizable. It would be unable to detect AprilTags in darker environments, where the tag's patterns are not as visible.

Fail Cases: The system will be unable to accurately detect tags that are not within the AprilTag template folder, causing misidentification if the unknown tag is similar enough to another. This extends to objects that look like AprilTags, but are not. If there is a pattern that is similar enough to a real AprilTag, it will be detected as that AprilTag.

SLAM Navigator (Em)





Accomplished: The navigator can successfully generate paths to specific frontiers and to given target AprilTags. It can get to a target AprilTag in several ways: a path directly to the tag (if location and environment are known), path plan to frontiers in the direction of the target AprilTag until it reaches the actual AprilTag (location is known, but environment is unknown), and through pure exploration (if location and environment are unknown). When searching, the navigator also does a scan of its surroundings to ensure that it sees all of the walls in the area to ensure the detection of all AprilTags. The navigator is also able to avoid collisions by preventing the robot from getting too close to an obstacle, backing up if the robot does get too close. The navigator is also able to visualize the path planning process, the tag locations, and the obstacle locations.

Limitations: If a goal location for a tag or frontier is considered unreachable or unavailable (inside a wall), the robot will not be able to path plan there. This system also performs better in smaller environments, as larger ones will take much longer to explore and find the given AprilTag locations. The environment must also be static to accurately path plan. The navigator is also set up so that if the environment is known, then all AprilTags there would have also been detected. This means that the robot will not search in a known environment for a target AprilTag.

Fail Cases: If there are no obstacles in the environment, the robot will be unable to use LiDAR to map the environment and localize itself, preventing the robot from being able to path plan to a target location. The actual LiDAR on the robot is able to malfunction, causing ghost walls to appear in the environment, preventing the robot from path planning through the "obstacles".

Conclusion

Overall, the project accomplishes all parts of the initial proposal. It is able to receive commands from people using gesture recognition, send those commands to the navigation system, which directs the robot, use SLAM to localize and map the environment, and detect AprilTags through the robot's camera. Any improvement on this project would involve updating and fixing the failure cases as well as addressing limitations for each part of the project, such as developing a

more versatile gesture detection system, adding a timer visual when a gesture is on screen, making the AprilTag detector more exclusive, and preventing the LiDAR from creating ghost walls.