Search and Pursuit Evasion

Mitchell Hornak, Alex Melnick, Maura Mulligan, Megha Shah

Github: https://github.com/alexrmelnick/EC545_Final_Project

Introduction and Goal

Search and Pursuit-Evasion is an ongoing research area in robotics where a robot needs to plan and react in an adversarial environment. An adversarial environment is one where the evader has elevated knowledge of the environment and is working against the robot. Relevant applications include: search-and-rescue operations for missing persons, natural disaster response, pursuit of criminals, animal herding, modeling animal behavior, and military interception of drones or vehicles in unwanted areas.

Our project goal was to program two robots to perform Search and Pursuit-Evasion. Using off-the-shelf Yahboom ROSMASTER X3 robots running ROS1 Melodic, a fully-autonomous pursuer robot and a semi-autonomous evader robot were programmed. In doing so, the team utilized multi-sensor integration and gained hands-on experience with industry standard tools such as ROS.

Design Diagram

The design of the system was done in parallel. It was designed leveraging ROS nodes and topics to run directly on the robots, and it was also designed in MATLAB/Simulink to test and demonstrate the finite state machine design and logic.

Evader

The evader robot is a human controlled robot with a goal of avoiding being captured by the autonomous pursuer bot. It consists of a logic controller that allows for human control unless there is an obstacle in front of it; in which case obstacle avoidance takes priority and the robot maneuvers to avoid a collision. The evader observes its surroundings through the use of LiDAR, while subscribing to the instructions for gameplay and the keyboard or joystick input from the user. The three topics published to the evader are raw LiDAR data in an array called ‘scan’, a boolean called ‘evader’, and a twist called ‘key_input’. All three of these topics are published to the control node called ‘laser_Avoidance’. This control node then determines the velocity and direction to send to the mecanum wheels through a twist topic, ‘cmd_vel’, and notifies the user when object avoidance protocol is taking over control by sounding a buzzer as a boolean over the ‘Buzzer’ topic. The mapping of the nodes and topics can be seen below in Figure 1.

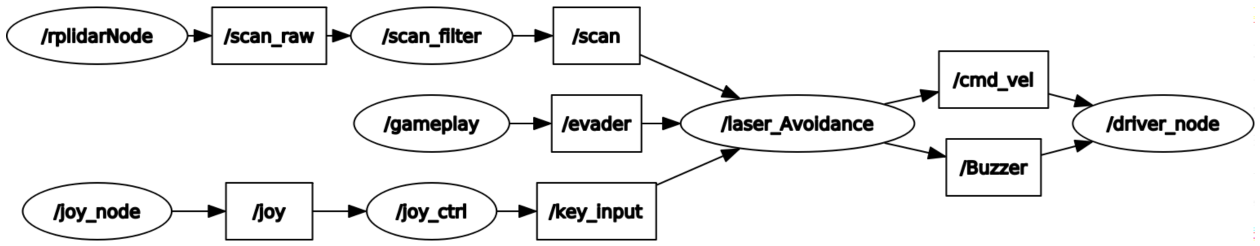

Figure 1: ROS rqt_graph of the Evader Package

The ‘laser_Avoidance’ node logic is as follows: first, check if it is in an active round (more details on the structure of the game are described later in the Gameplay section). The evader can only move during an active round and five seconds prior to the start. This is determined by a boolean signal sent from ‘gameplay’. Next, process the LiDAR data, ‘scan’, to check if there is an obstacle in the 60 degree front-facing horizontal field of view. If an obstacle is present, determine if it is to the left, right, or center, then send the respective rotational velocity about the z-axis to turn the robot away from the obstacle with the shortest path possible. Throughout this process, a buzzer is enabled to notify the user that an object was identified and their control is temporarily suspended. Finally, given it is an active round and there are no obstacles in the path, give control to the human operator by feeding the ‘key_input’ directly to the ‘cmd_vel’. The task priority of the evader robot is as follows: gameplay management, obstacle avoidance, and then user control.

Pursuer

The pursuer robot is fully autonomous, its goal being to capture the evader robot within a 0.5 meter capture radius (plus or minus 10cm). The pursuer utilizes sensor fusion of LiDAR and depth camera data to accurately navigate the arena. The primary sensor for pursuit and tracking is the depth and color cameras, while the LiDAR acts as an obstacle avoidance safety layer. The depth image nodes, shown within the large box in Figure 2, publish the depth image data as the `image_raw` topic. The ‘colorHSV’ node processes the color images from the camera and identifies the color we specify by hue, saturation, and value, in this case we chose red, hence why the evader is covered in red tape. Once red is identified, ‘colorHSV’ will pinpoint the relative center of the red area and estimate a radius to draw a circle around the region, as seen in Figure 6. ‘colorHSV’ will publish to the ‘Current_point’ topic the center point as coordinates in X and Y values that translate to the pixel location of the color from the image and also publish the drawn circle radius.

The pursuer robot is primarily controlled by the `colorTracker` node, which subscribes to the `image_raw` topic and ‘Current_point’ topic from the ‘colorHSV’ node. If there is a positive, non-zero radius from ‘Current_point’, the robot knows it has detected the color red and begins pursuit. ‘colorTracker’ matches the X and Y pixel coordinates of the tracked object to the corresponding pixels from the depth frame to calculate the pursuer's distance from the evader, then uses a PID to calculate linear velocity. ‘colorTracker’ also uses a PID with the X coordinate of the color center to calculate the necessary angular velocity required to keep the color in the center of the frame.

The depth camera excels at identifying objects at distances greater than 0.5 meters, but struggles with objects closer than this, often returning erroneous results: this is where LiDAR comes in. The `colorTracker` node is also subscribed to the `Laser_Dist` topic, which is a boolean. The `laser_distance` node processes LiDAR data, identifies if an object is within the central 60 degrees of its forward facing horizontal field of view, and publishes True if there is an object within 0.5 meters, and False if there are no obstacles. When True is published, the pursuer’s obstacle avoidance takes over, being prioritized over any color tracking, and it drives straight backwards until it is at least 0.5 meters away from the obstacle, it is then able to return to the search and pursuit modes.

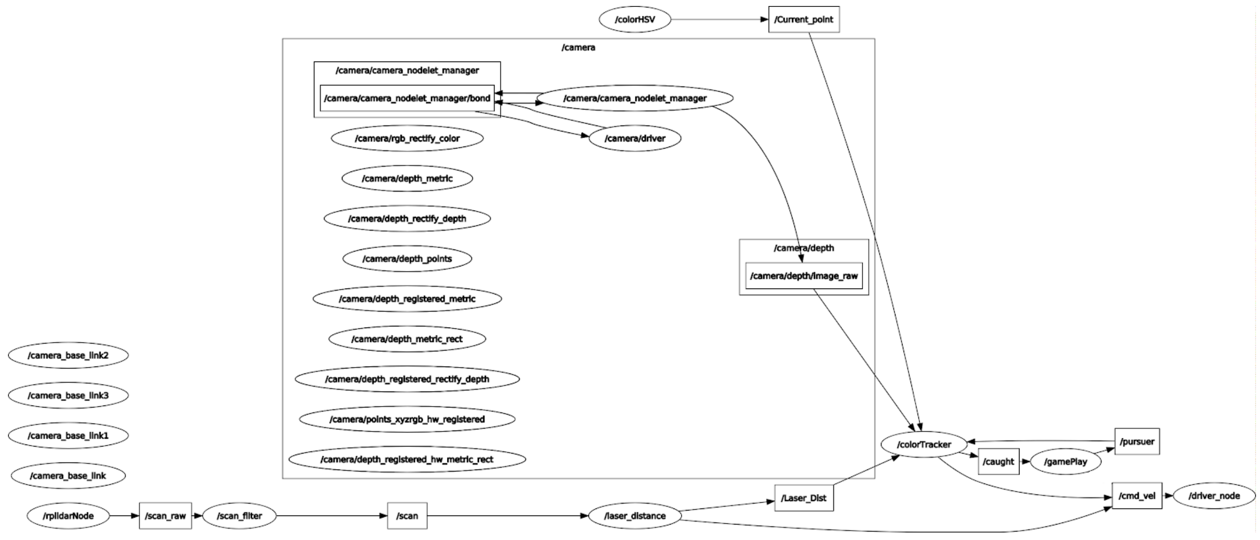
/colorHSV
/Current_point
/camera
/camera/camera_nodelet_manager
/camera/camera_nodelet_manager/bond
/camera/camera_nodelet_manager
/camera/driver
/camera/rgb_rectify_color
/camera/depth_metric
/camera/depth_rectify_depth
/camera/depth_points
/camera/depth_registered_metric
/camera/depth_metric_rect
/camera/depth_registered_rectify_depth
/camera/points_xyzrgb_hw_registered
/camera/depth_registered_hw_metric_rect
/camera/depth
/camera/depth/image_raw
/camera_base_link2
/camera_base_link3
/camera_base_link1
/camera_base_link
/rplidarNode
/scan_raw
/scan_filter
/scan
/laser_distance
/Laser_Dist
/colorTracker
/caught
/gamePlay
/pursuer
/cmd_vel
/driver_node

Figure 2: ROS rqt_graph of the Pursuer Package

The pursuer robot has multiple states it can enter, each of which is represented by illuminating the Yahboom LED light strip a certain color, to provide quick and accurate debugging. The search state, represented by yellow LEDs, is when there is no red color detected, causing the pursuer to rotate continuously until it detects the evader bot (or any object within the specified HSV range). Once the evader is located, the pursuer bot transitions to the pursuit state, represented by red LEDs, and drives toward the evader using the PID control previously described. When the pursuer captures the evader, it enters a captured state, represented by blue LEDs, where it pauses for 10 seconds to give the evader time to reset and strategize before resuming the chase. If the pursuer is able to outlast the evader for 30 seconds, purple LEDs illuminate and the gameplay node pauses the pursuer, signifying an evader round victory. Finally, there is our low level LiDAR obstacle avoidance state, represented with green LEDs, which illuminate when the robot is backing up because of an obstacle detected in front of it that is not the evader. The task priority of the evader robot is as follows: gameplay management, obstacle avoidance, and then autonomous color tracking.

Gameplay

A game was implemented to constrain the pursuit in terms of time and allow for multiple trials. The design of such in ROS can be seen in Figure 3. The game is a series of rounds that can last up to 30 seconds each. The round ends when a boolean ‘caught’ signal is sent to the ‘gameplay’ controller. This signal is true if the pursuer caught the evader (within half a meter), or false if 30 seconds has elapsed and the evader has not been caught. At the end of the round, both robots are disabled for five seconds using the ‘pursuer’ and ‘evader’ topics to disable all motion. During this time, the score is updated - incrementing the pursuer if ‘caught’ is true, and incrementing the evader if ‘caught’ is false - and published to the ‘points’ topic to be displayed in the terminal through echoing the rostopic. The evader is then re-enabled and given a ten second head start before the pursuer is also re-enabled, at which point the next round begins. This process is repeated until a ‘reset’ boolean is published that resets both the scores to zero.

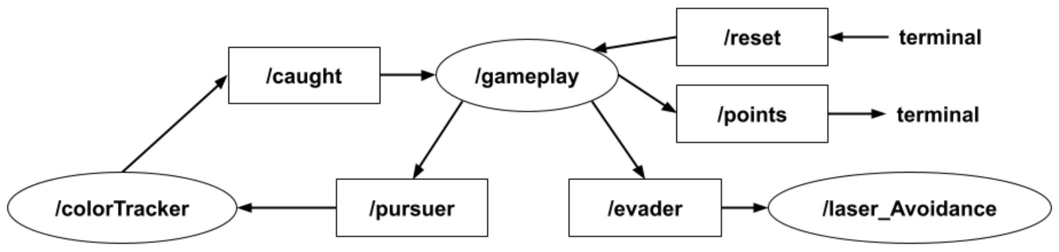

Figure 3: ROS rqt_graph of Gameplay with Multi-Agent Communication

The gameplay is designed to talk to both the pursuer and evader. Ideally this would be hosted on a shared ROSMASTER_URI but the current implementation only has it communicating with the pursuer. The challenges that resulted in this are highlighted below in the Technical Challenges and Mitigations section. That being said, the only observable difference is that the evader is always in an active state where it can be controlled (and perform obstacle avoidance). Otherwise the scorekeeping is still accurate and the pursuer toggles between an inactive and active state.

Specifications and Modeling

To demonstrate the successful design of the system, it was executed alongside a list of specifications and a parallel model was explored in simulation. The specifications can be seen below in Table 1, and are broken into three subcategories: the evader, the pursuer, and gameplay. Although some of the implementation had to be adjusted due to technical challenges, all were still accomplished. One thing to note: the 5 second pause of the evader in gameplay that occurs at the end of each round must be done manually by the user as multi-agent communication was not possible given the timeline.

Table 1: List of Specifications and Completion Status

Specification	Status
Human-Controlled Robot (Evader)	✓
● LiDAR obstacle avoidance overtakes human controller when within specified collision range	✓
● Buzzer sounds when obstacle avoidance takes control	✓
● User can remotely maneuver the robot in the xy-plane with keyboard inputs	✓
Autonomous Tracking Robot (Pursuer)	✓
● LiDAR obstacle avoidance initiates reverse command when within 0.5m of an obstacle	✓
● Robot will spin to search for target when nothing detected	✓
● Robot will track and pursue the evader when its detected, specifically looking for the color red	✓
● LED light strip displays unique color corresponding to state of the pursuit	✓
Gameplay in the Arena	✓
● Autonomous robot pursues human controlled robot	✓
● When pursuer robot captures evader robot, pursuer robot pauses for 15s, evader robot pauses for 5s, timer resets, and win counter ticks up one point for the pursuer	✓
● When evader robot evades pursuer robot for 30 seconds, pursuer robot pauses for 15s, evader robot pauses for 5s, timer resets, and and win counter ticks up one point for the evader	✓
● The user can send a reset command to reset the game scores to zero	✓

These specifications being met were demonstrated on the physical robots in an arena, and also match the model in simulation. Figure 4 below displays the preemptive hierarchical state machine of the overall system. At the higher level, the “Game ON” state transitions to “Game OFF” as soon as the caught signal is received or the 30s timer expires. Since it is a preemptive transition, the external transition takes precedence over internal transitions and thus internal transitions are not executed when “Game ON” state is preempted. In the “Game OFF” mode, the scores are updated and a 5 second timer runs before transitioning back to “Game ON”. The “Game ON” state is an asynchronous concurrent composition of the pursuer and evader FSM. Asynchronous concurrent composition means that the component state machines react independently. The transition between states is controlled by events occurring, in this case the presence of an obstacle or not. The pursuer includes a tracking state that includes searching, tracking, and catching the evader. It switches to an obstacle avoidance state in the presence of an obstacle and vice versa. Similarly, in the evader FSM, the obstacle avoidance state overtakes human control upon detection of an obstacle and yields to human control state once the obstacle is averted. The pursuer can be in a tracking state while the evader could be in human control or obstacle avoidance state. Purser and evader states do not necessarily have to react simultaneously.

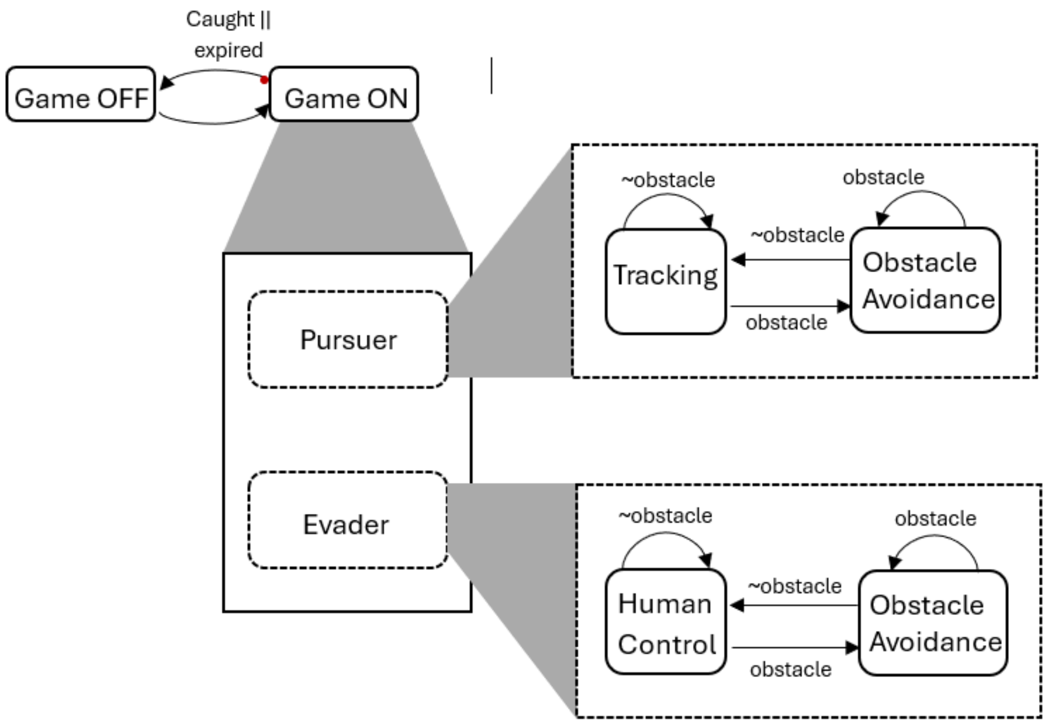

Figure 4: Hierarchical State Machine of the Pursuit-Evasion System

The state machine was modeled in MATLAB\Simulink using Stateflow. An example of the simulation output can be seen in Figure 5. As future work, designing the simulation to run in real time would be ideal so that the tracking, obstacle avoidance, capturing, and game timers could be visualized in higher fidelity.

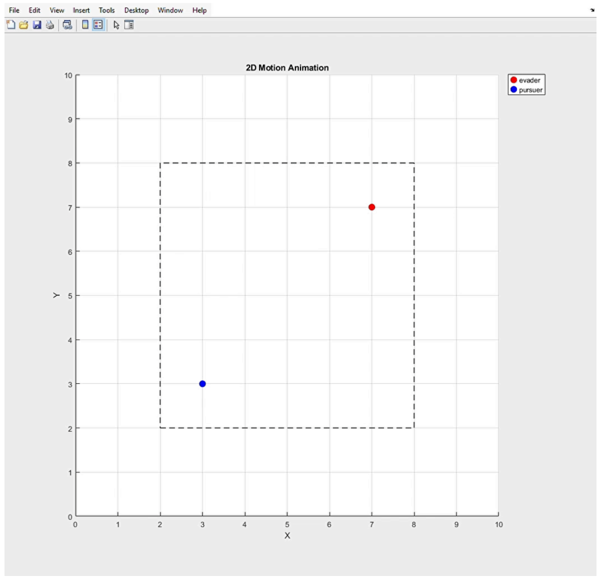

Figure 5: Animation Showing a Simulation of the Finite State Machine

It is important to note that the model is a simplified representation of a real-time robot pursuit. It captures only key aspects and may not include all the nuances and unpredictable elements present in the physical execution such as human control, sensor variation, and side effects from integration like latency in communication. However, it is a great approximation to visualize overall implementation and fundamental workings of a complex system.

Hardware/Software Implementation

The pursuit-evasion game was implemented on two Yahboom ROSMASTER X3 robots that have a LiDAR unit, depth and color cameras, mecanum wheels, LEDs, a buzzer, and a Jetson Nano onboard. The game was executed in a makeshift arena with cardboard walls to contain the game with the LiDAR obstacle avoidance, as seen in Figure 7. Both robots operated independently of one another: one as the human-controlled evader, and the other as the autonomous pursuer. The robots run ROS1 Melodic and Python 2.7 which are the only softwares used in the implementation. Specifically, the pursuer robot leveraged OpenCV for the color tracking algorithm.

The logic was programmed onboard through the nodes and topics discussed above in the Design section. To run them, a secure shell (ssh) was used to communicate with both and operate them in a head-less mode. This ensured all computation was done onboard which minimized the latency and required bandwidth by disabling the stream of camera images from the color tracking. Regarding the evader, ssh was also the method used to gather keyboard input to control and steer the robot. An additional launch had the use of a joystick in place of the keyboard which utilized a standard video game controller that wirelessly connected to the robot. However, due to limitations in the wireless connection distance, the keyboard control ended up being more reliable and robust to distance.

Performance Analysis & Optimization

Extensive real time testing was performed with the two robots operating in our cardboard arena to analyze performance, debug the system, and ensure we were meeting specifications. As discussed in the Pursuer section, we associated each possible state of the pursuer with an LED color to ensure that our state machine was entering the right states at the right times. A similar system was used with the buzzer of the evader robot, where the buzzer would sound when the LiDAR obstacle avoidance took control so we knew that it was performing properly. Additionally, during testing, we had our 'colorTracker' ROS node printing all relevant information to the terminal such as the LiDAR boolean, measured distance from evader, as well as linear and angular velocity, to ensure our algorithms were running properly.

The first problem we encountered with the pursuer robot performance was the linear velocity would rapidly change, often jumping too high before dropping down to a slow crawl. Upon further investigation, we discovered that the perceived distance to the evader was highly sporadic, which was getting sent into our PID, causing the jumps in velocity. These distance measurements were being affected by the colorHSV node frequently picking up background noise in the environment that it perceived to be the color of interest. This error prevented our pursuer robot from meeting specifications because it was not accurately tracking our evader. We solved this problem by painstakingly tuning the HSV parameters the node was looking for to reduce as much noise as possible. The result of this tuning can be seen on the right of Figure 6, where the white is all the red in the image being observed. Another layer of safety we added was a running average for the distance measurements, which keeps track of the 5 previous observed distances. If the incoming distance observed is substantially larger or smaller than our running average, we throw it out and consider it erroneous, then continue velocity control based on our average. Another mitigation strategy we implemented to reduce false tracking was limiting the field of view of the color tracker camera. We noticed that most erroneous colors that tricked the tracker were in the top 50 pixels of the camera frame, usually due to people wearing red or other red background objects, so we modified the color tracking algorithm to ignore all instances of red seen in the top 50 image pixels. One final strategy was to create the cardboard arena wall as seen on demo day in Figure 7, which abstracted the environment by removing any potential background red colors. All of these improvements to our system resulted in the smooth pursuer linear velocity we saw on demo day (additional video linked below).

The next issue we encountered was the real time performance of the pursuer. We observed that there was a substantial delay between what the pursuer perceives and when it reacts with velocity updates or LED changes to represent different states. This interfered with us achieving our specifications because the state of the gameplay was unclear, the evader could pass the pursuer and think they escaped but seconds later the capture state would be processed and end the round even though the evader was now out of capture radius. We knew that we needed better real time performance to create a realistic pursuit-evasion scenario. Due to the nature of this issue, we quickly realized that this bottleneck was a result of poor scheduling so we opened our code to investigate the ROS rates and queue sizes of our publishers and subscribers. Our first realization was that queue sizes were either set at 10 or not set at all, so we reduced them all to one to process only the most recent data, even if some values were dropped it didn't matter because we only need the most current data. Next, we investigated our ROS rate, which was initially set at 20Hz. After conducting some quick research we found that real time robot control with ROS should be between 5-50Hz. We deduced that either 20Hz was too high and our processor could not keep up with the computations or the rate was too low and velocity was not being updated often enough. We dropped the ROS rate to 5 and 10 in two separate trials and noticed nominally poorer performance, so we then increased the rate to 50Hz and witnessed a substantial improvement in real time tracking by the pursuer. With these two adjustments to our system we achieved accurate, real time tracking and capture of the evader robot. These fixes actually gave the pursuer such an unfair advantage that we had to slash its max speed just to give the evader a chance at winning a round of the game.

Technical Challenges and Mitigations

Throughout development, we encountered a series of unexpected technical challenges that required us to adjust our final goal. However, mitigation plans were put into place when possible to execute the demonstration successfully. These mitigation strategies primarily consisted of abstracting the environment or adjusting how the logic was being implemented given the time constraints.

Multi-Agent Communication

An initial goal was to have both the pursuer and evader communicate the gameplay going on. The positions and all action would be exclusive and hidden from one another so as to not give one an advantage, only game points and the state of a round would be broadcast. This multi-agent communication can be seen in the Gameplay section. While it is not necessary to have both robots talk to run the game, it was a stretch goal to gain experience with multi-agent communication.

Both the pursuer and evader were able to successfully host one another on their ROSMASTER_URI and could run their respective packages with the master being on the other robot. However, simultaneous execution of both the evader and pursuer package on the same ROS master caused significant issues. Specifically, there were node and topic name clashes as both used the same nodes and topics to physically drive and manipulate the vehicle. To combat this, grouping and identifying namespaces for each robot in the launch file was attempted. While this is believed to still be a possible solution, due to time constraints and inexperience with ROS it was unable to be debugged and properly tested in time.

Latency Issues

Latency was experienced on both the evader and pursuer and materialized as either a delay in the maneuvering of the robots or state updates. The control of the evader was accomplished through keyboard input on a secure shell, which resulted in a slight delay due to the nature of wirelessly connecting the robot to a laptop. An additional attempt was made to run the evader entirely on-board with the provided wireless joystick. This solution, however, posed its own challenges with an extremely limited joystick range which caused frequent disconnection from the robot. Therefore, the slight delay of the secure shell was accepted as the keyboard control was more robust and reliable which was of higher priority.

Regarding the pursuer, much time was dedicated to ensure running the ROS package remotely was occurring in a completely head-less mode. Given the reliance on OpenCV and the computationally expensive image processing required, any stream or display of the frames would waste bandwidth and cause latency. To mitigate this problem we disabled all video streams to our connected computer (which we had running for debugging purposes), which allowed our evader to focus solely on receiving commands and our pursuer to dedicate more computational power to image processing and color tracking. Efforts were made to further minimize latency, but it was impossible to have a near real-time response given the resources, so some slight delay was accepted in the final system because it did not interfere with our specifications.

Tracking Incorrect Background Objects

The object tracking algorithm primarily relied on color as its main source of information, which came with distinct advantages and disadvantages. The biggest strength—and weakness—of this approach is its simplicity. Color tracking is implemented by prescribing a range of hue, saturation, and value (brightness) of the desired color we want to track and then searching the observed color image for groups of pixels within this range. While the most obvious method might involve using the RGB color space, HSV is widely regarded as more accurate for robotic color tracking purposes. This approach is significantly simpler and easier to understand compared to complex machine vision AI algorithms. However, it is prone to tracking incorrect objects. Since the algorithm locks onto a specific color rather than the object itself, any other objects in the field of view that share the same color can also be mistakenly tracked. This issue occurred frequently during testing and our mitigation strategies for this issue were discussed in the previous section.

Demo Day Error State

During demo day, an error state was witnessed where the pursuer incorrectly entered the captured state, turning the LED lights blue when Professor Li stepped between the pursuer robot and the evader robot to test the LiDAR obstacle avoidance. An analysis on the code was conducted to hypothesize where the fault arose and how the state machine logic could be improved to eliminate this error, given more time. In the left frame of Figure 6, the colorHSV node identifies the color of interest, draws a circle around the colored area, and pinpoints the center of the circle. The colorTracker node receives X and Y pixel coordinates of the circle center in the HSV image and matches it to the corresponding pixel from the depth camera frame to assess the distance to the evader robot. To increase positional accuracy and reduce the impact of erroneous sensor readings, the colorTracker node averages the depth points that are plus or minus three pixels in the X and Y directions with the center depth.

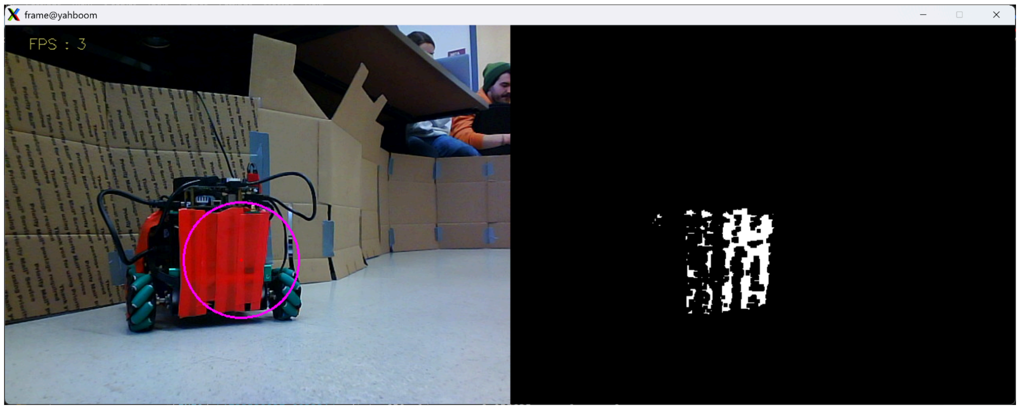

Figure 6: Visualization of Algorithm Identifying Red HSV Values

It’s hypothesized that when the professor stepped between the robots, the pursuer still had vision of the evader, but when calculating average distance, a few depth points ended up capturing the professor’s legs, which caused the final distance result to average to the capture range of 0.5-0.6m, tricking the pursuer to think it has captured the evader. This was an error state that we had not considered because not much testing was performed with antagonistic agents in the arena, since the professor was deliberately attempting to trigger the obstacle avoidance while the pursuer was tracking.

This further emphasizes the need for extensive system testing as a part of the engineering design process, since other people can discover new ways your system can fail that had not previously been considered.

Unfortunately, the solution is not as simple as eliminating the depth camera pixel averaging. This was attempted early in the design process and testing showed that due to variations in lighting, environment, and size of the target (due to distance away), the color tracking algorithm frequently identified the edges of the colored regions of interest, not always the center. When these edge pixels are correlated to the depth frame, there is a high probability of the depth frame identifying that pixel as the wall behind the target, which grossly overestimates the distance to the target, causing inaccurate linear speed calculations from the PID. The more efficient way to mitigate this problem would be to implement YOLO or a more accurate color tracking algorithm that is able to draw a border around the colored region detected, not just a circle, since the detected area will not always be well represented by a circle. We would then be able to take our depth frame average from a random selection of pixels within the drawn border, eliminating false positives when unknown agents are within the arena. Before pursuing this idea, though, we would need to perform an investigation into whether the Yahboom has the computational power to perform more complex image processing.

Division of Labor

The high level breakdown of what each individual team member worked on can be seen in Table 2. Everyone worked together on the integration of the individual parts and debugging the system. Each team member became very familiar with ROS and the construction of custom packages.

Table 2: Team Members and Individual Contributions

Member Name	Contribution
Mitchell Hornak	Pursuer robot logic design, testing, debugging, and implementation. Full system testing and debugging
Alex Melnick	Pursuer robot logic design, testing, debugging, and implementation. Full system testing and debugging
Maura Mulligan	Evader robot design and implementation, gameplay design and implementation, full system integration and debugging
Megha Shah	Evader robot design and implementation, modeling and simulation, full system integration and debugging

Conclusion

This project successfully met its objectives, showcasing a robust implementation of a Search and Pursuit-Evasion system. The autonomous pursuer robot demonstrated its ability to track the evader bot, avoid obstacles, and navigate effectively within the arena using advanced sensor fusion of LiDAR and depth camera data. The evader bot operated seamlessly, allowing for manual control with low latency while maintaining its ability to autonomously avoid obstacles when necessary.

The gameplay logic functioned as intended, enabling accurate tracking of game points and ensuring the pursuer robot paused appropriately upon capturing the evader. Additionally, the system design highlighted the effective use of industry-standard tools such as ROS1 Melodic and OpenCV in Python, providing team members with hands-on experience in developing multi-agent systems.

Despite some technical challenges encountered during testing and demo day, the team demonstrated adaptability and problem-solving skills to deliver a functional and reliable system. This project not only reinforced an understanding of multi-sensor integration, finite state machines, and real-time robotics systems, but also underscored the importance of collaboration and iterative development.

Overall, this pursuit-evasion system represents a meaningful step toward applying robotics in adversarial and dynamic environments, with potential implications for search-and-rescue, autonomous navigation, and similar fields.

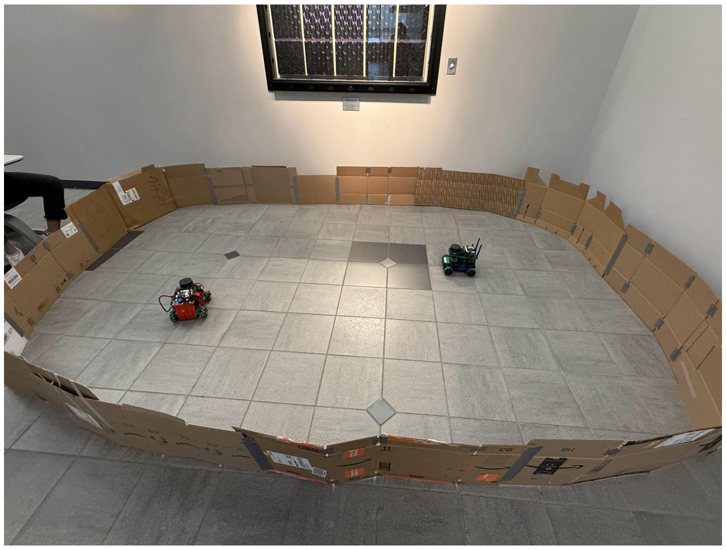

Figure 7: Final Arena Setup with Both Pursuer and Evader Robots

Demo Day Video Link (videos will also be shared with Professor Li on Google Drive):

https://drive.google.com/file/d/1DgFXjcAWtT1CfOkr8_2-Z2fz_jshTBSl/view?usp=sharing