Background and purpose

The MEMS1097 final project is a robotics delivery task using two robots: the PX-100 pincher robot and the TurtleBot3 Burger. The goal of the project is to show a multi-robot system with basic manipulation, vision, and navigation working together. The project has two main stages. First, the PX-100 pincher picks up a highlighter from the ground and hands it to the TurtleBot. Then, the TurtleBot drives through a cardboard maze and delivers the highlighter to a drop-off location at the exit.

For our extension to the project, the TurtleBot uses a CS90 Logitech camera and an ArUco tag to navigate to a marked garage area after exiting the maze. This extension demonstrates simple computer vision positioning. This document describes how we set up our robots and how each task was accomplished.

Besides the robots, our project also consists of other pieces. We used two provided Logitech cameras. One camera was used for the pincher, which was mounted on top the tip of the wooden hanger that we built, as shown in Figure 1. This will allow the camera to view the entire pincher from the top and us to measure the distance from the camera to the pincher base.

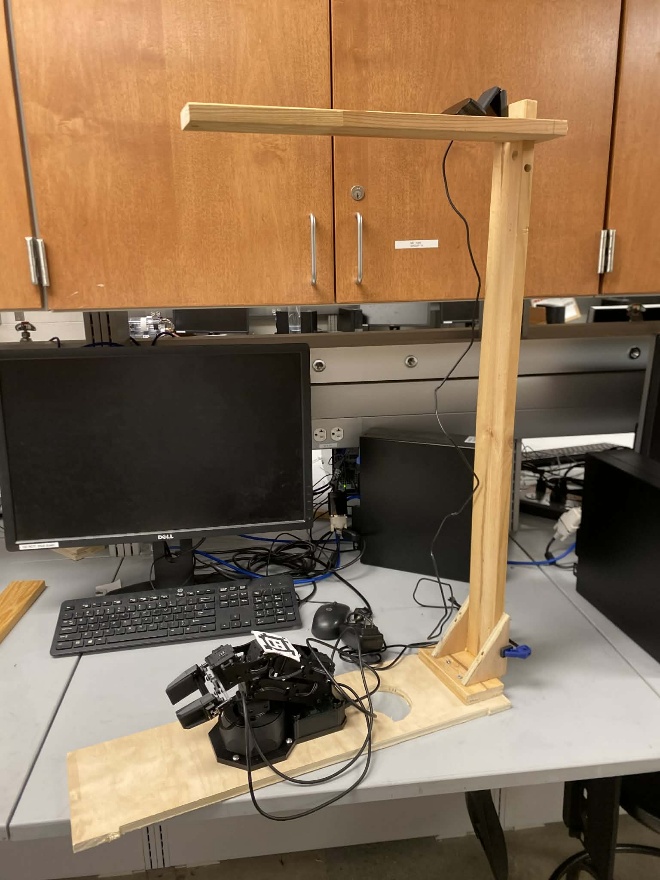


Figure 1: Pincher Robot Setup

The second Logitech camera was connected to the blue USB port of the TurtleBot Raspberry PI and mounted in the front of the TurtleBot just below the Lidar, as shown in Figure 2. This allows the camera to detect the ArUco tag at the exit without obstructing the Lidar. Also shown in Figure 2 from the back of the TurtleBot is our 3D printed highlighter release mechanism. We have a small box to hold the highlighter; the bottom of the box is attached to a servo motor. When the motor rotates, it will drop off the highlighter.

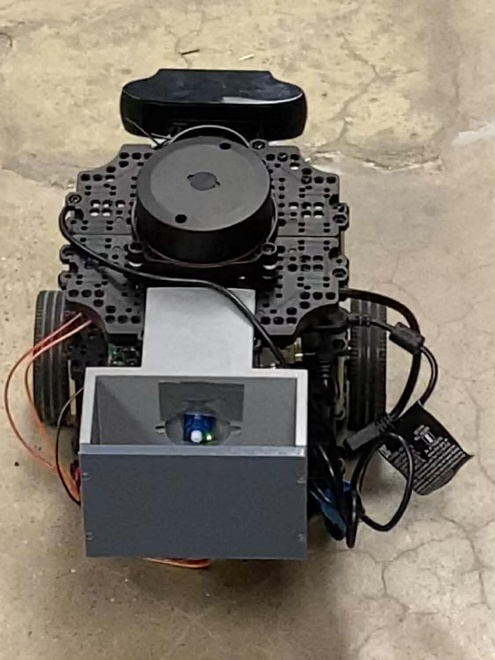


Figure 2: TurtleBot Setup

Highlighter Pick and Place

In the first stage where the pincher picks up the highlighter, we built a simple vision guided pick and place system that lets the PX-100 automatically grab a highlighter and drop it a high position into our 3D printed box. The system is split into two main parts, a camera node that converts pixels into a 3D target point in the pincher base frame, and a control node that takes subscribes to that target point and runs a short pick and place sequence using interbotix API and its built-in inverse kinematics or IK solver. We thought about making our own IK solver but decided to use the Interbotix API instead. The built-in solver is already tuned to the PX-100 model for geometry and joint limits, so it is less likely to produce unreliable or strange configurations. It also lets us command poses directly in x-y-z space using the *set\_ee\_pose\_compoenents*, which is one of the functions that matches naturally with 3-D target points from our camera. In other words, we just have to tell the node using the library where we want the gripper to be in the base frame and *set\_ee\_pose\_compoenents* calls the IK solver to find the joint angles. This avoids us having to manually compute or tune our joint positions and makes this task much simpler and more reliable.

A CS90 Logitech USB camera is mounted above the PX-100 as shown in the previous section, looking down at the arm and the highlighter. A camera driver node publishes the raw images on /image\_raw with calibration data from /camera\_info. These images are processed by a blob-detection node that uses tunable HSV thresholds to determine the blob color that we want, this is blob detection is identical to Lab 2 of this class. This node isolates the highlighter, cleans up the image, and finds the largest contour. From this contour it computes the image centroid and publishes it as a geometry\_msgs/PointStamped on /blob\_px, where point.x, point.y are the pixel coordinates in the camera frame.

A second node, blob\_ray\_plane\_node, subscribes to /blob\_px and /camera\_info, uses the camera intrinsics and the known transform from the camera frame to base\_link (which we measured) to cast a 3-D ray through the pixel, and intersects that ray with the table plane. The output of this is a 3-D point of the highlighter on the table expressed in the robot base frame, published as /blob\_point\_base.

We have found that the camera runs a lot better on a separate computer, which can help with the performance of our robot. Therefore, the camera driver and pose estimation node were run on a different computer instead of our laptops. When we ran the camera node inside the ROS2 VM on our laptops, the frame rate was low and the images lagged, which made the blob position jumpy or delayed. Moving the camera node to a separated machine with direct USB access gives much faster image processing. ROS2 communication is over the network, the topics /image\_raw, /camera\_info, and /blob\_point\_base can be used as long as all the machines share the same ROS\_DOMAIN\_ID. From the rest of the system’s point of view, it doesn’t matter which computer actually publishes the camera data, it just sees a faster camera pose. Figure 3 shows the computer we used for the camera node.

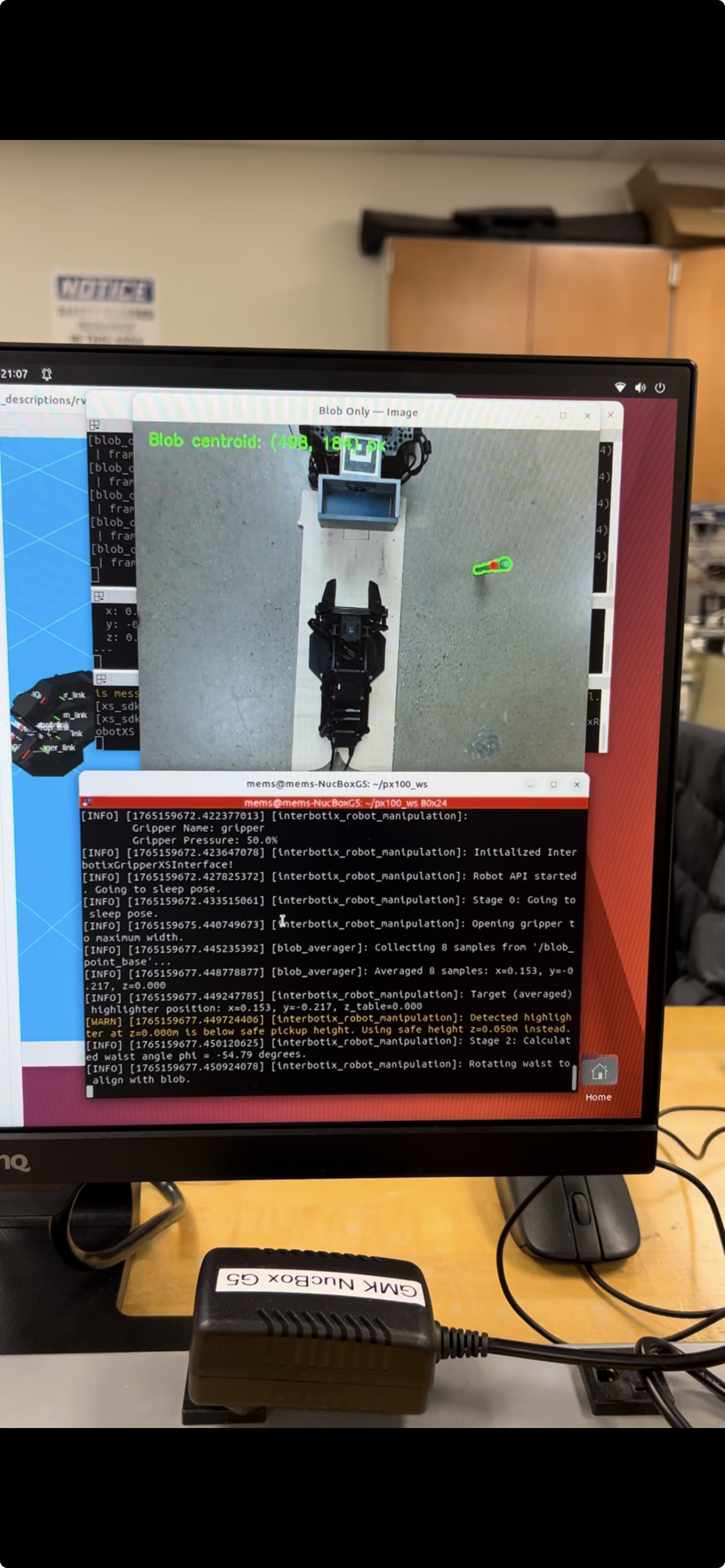


Figure 3: Separate Computer for Camera Nodes

On the robot side, we used the Interbotix “InterbotixManipulatorXS” Python API to control the PX-100. This API uses PX-100 manufacturer’s internal IK solver, so as mentioned above, instead of computing joint angles ourselves, we can command the end-effector or the gripper poses directly using set\_ee\_components(cartesian system arguments), and the library finds joint positions that within the arm’s limits. Our pick-and-place node subscribes to /blob\_point\_base, collects some samples and averages them to reduce noises. It then runs a finite state machine with 10 states:

1. Go to sleep pose, gripper open to max width.
2. Wait for several /blob\_point\_base samples, average them.
3. Calculate waist angle phi to point directly at blob centroid.
4. Rotate waist to align with blob.
5. Extend arm straight out until blob is between grippers (at safe pickup height).
6. Close gripper to grasp highlighter.
7. Lift to drop height.
8. Move to drop target location.
9. Open gripper fully to release.
10. Return to sleep pose.

Overall, the logic is: the camera continuously converts the highlighter’s color (green) blob into a 3D point in the robot base frame, and the PX-100 subscribes to that point and uses the interbotix IK API to execute a repeatable pick and place motion. The problem with this setup is that the calibration of the camera frame to the base frame can be disrupted if the camera position or angle is changed by accidentally bumping it. That’s why the pincher couldn’t pick up the highlighter during the final demo. This can be fixed by manually adjusting the camera until the calibration is back to normal, but that can take a few iterations.

TurtleBot maze navigation and ArUco parking

After the PX-100 pincher releases the highlighter, the TurtleBot3 Burger transports the highlighter through the maze and delivers it to the final depot. At the exit of the maze, it will detect the ArUco tag on one wall of the garages and navigate to that position, turn around and drop off. Figure 4 shows our foamcore garage.



Figure 4: Foamboard Parking Garage

As mentioned in the background section. A Logitech C920 USB camera was connected directly to the Raspberry PI on the TurtleBot using the blue USB 3.0 port. The camera was mounted on the front of the TurtleBot just below the LiDAR. The camera node runs directly on the Raspberry Pi. Camera calibration parameters are loaded through the /camera\_info topic so that pixel measurements can be converted to accurate pose estimates relative to the robot.

Before reaching the ArUco parking area, the TurtleBot navigates through a cardboard maze using a left-wall following node. This node relies on the LiDAR data to maintain a fixed distance from the left wall by adjusting the robot’s angular velocity while driving forward. Once the TurtleBot exits the maze and the ArUco marker becomes visible, the wall-following node is disabled and control is handed over to the parking controller.

To guide the TurtleBot to the correct drop-off location, an ArUco parking node was used. This node processes the camera's images, detects a single ArUco maker, and estimates its pose relative to the camera frame. The pose is then transformed into the TurtleBot base frame and published as PoseStamped message on the /aruco\_pose topic. We decided to use ArUco maker because it is reliable at providing repeatable position and more robust to lightning changes and background than any color base method. Our system was tested under both low-light (our room) and bright conditions (sub-basement). While the color-based blob detection used for the PX-100 arm was sensitive to changes in lighting and required retuning in different environments, the ArUco based detection on the TurtleBot remained reliable and consistent.

The parking controller node subscribes to the /aruco\_pose topic and generates velocity commands for the TurtleBot by publishing Twist message to /cmd\_vel. This node was used as a finite state machine with three states of operation. In the approach state, the robot drives forward while centering itself on the ArUco tag. Linear velocity is controlled based on the distance to the marker in the camera frame. Once the robot reaches the desired target distance, the controller transitions to a turning state, where the TurtleBot turns approximately 180 degrees using a fixed angular velocity for a specified duration. We found that depending on the surface of the ground, we needed to tune either velocity or time it rotates to turn it 180 degrees. After the turn is complete, the controller enters a done state, publishes zero velocity commands and stops the robot.

To complete the delivery task, a simple highlighter release mechanism was mounted on the back of the TurtleBot. A small 3D-printed box holds the highlighter, and the bottom of the box functions as a trap door driven by a small SG90 servo motor. The servo is controlled by a ROS 2 node running directly on the Raspberry Pi. This node uses the RPi.GPIO library to generate a 50 Hz PWM signal on a GPIO pin connected to the servo’s signal wire. When triggered, the servo rotates from approximately 0° to 180°, opening the trap door and allowing the highlighter to fall out. After the motion is complete, the PWM signal is stopped to prevent jitter. Figure 5 shows how we wired the servo to the TurtleBot PI.

A diagram of a wiring diagram

AI-generated content may be incorrect.

Figure 5: Servo Motor Wiring

Overall, the project successfully demonstrated a simple but complete multi-robot delivery system that combines computer vision, manipulation, and navigation as part of one continuous system. The PX-100 pincher used color-based blob detection and the Interbotix IK API to automatically pick up a highlighter and place it into the TurtleBot’s box, while the TurtleBot navigated the maze and used ArUco base pose to park in the garage and release a highlighter at the drop-off location. Although the system was a bit sensitive to calibration and lightning, the architecture of transforming camera measurements into base frame targets and letting controllers handle the motion proved robust and repeatable. For the future, we can improve the system by making camera-pincher calibration more automatic, adding better error response when grasp fails and integrating two robots, pincher and TurtleBot, into a single machine. These improvements will make delivery more reliable and applicable to a real-world application.