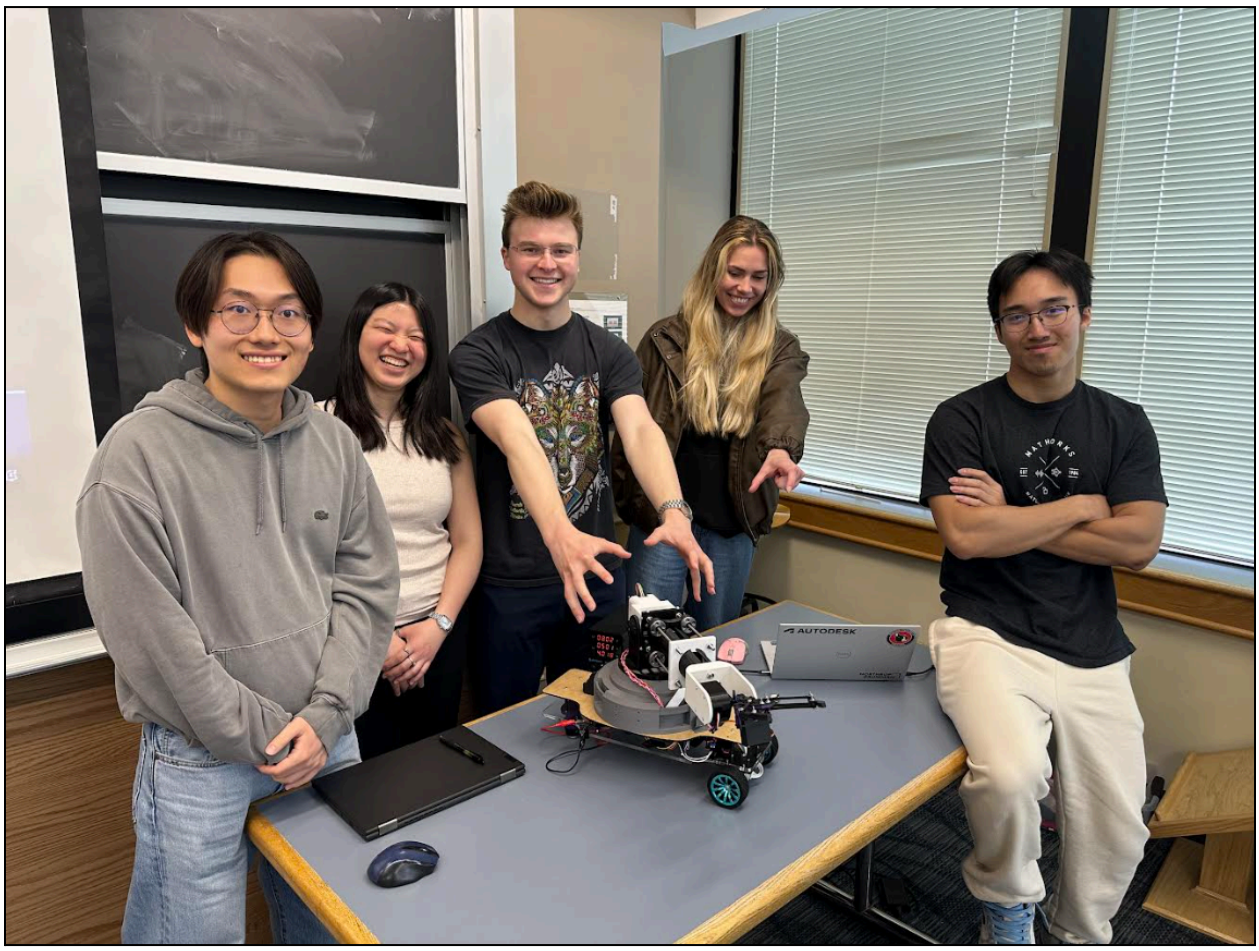
Final Report

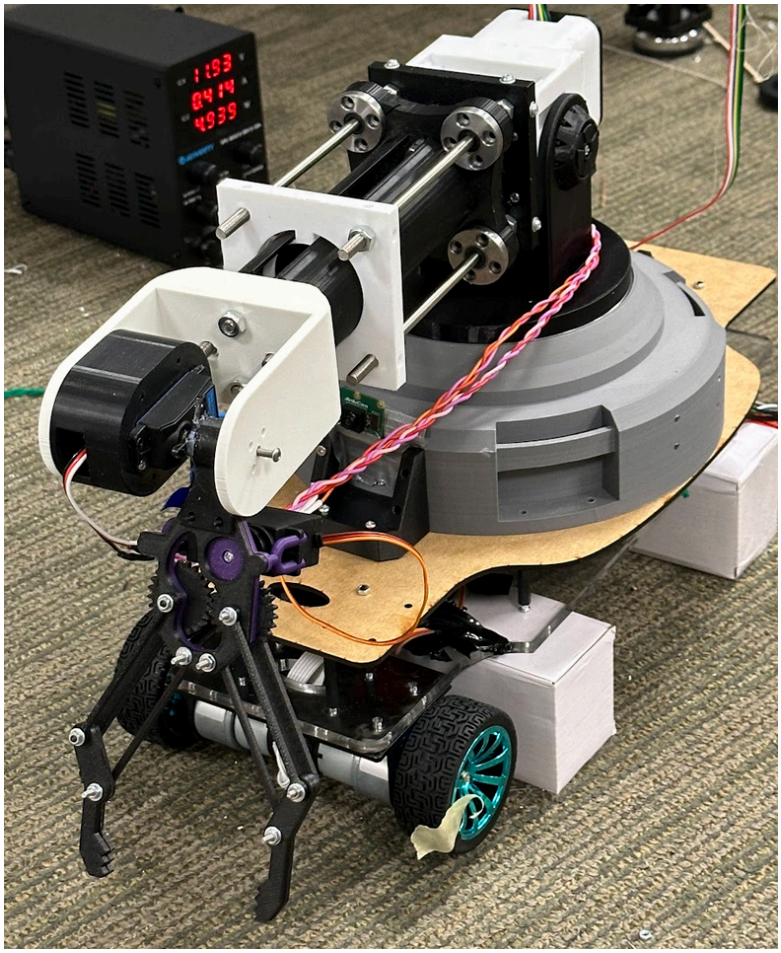
ME416 (Introduction to Robotics)

By Owen Jiang, Jason Rich, Ksenia Suglobova, Kimberly Wong, Justin Yu



Overview:

Our ankle-grabbing robot, referenced as Christopher, is a 5 DOF (3 DOF arm + 2 DOF wheels) robot that performs the task of autonomously grabbing the ankles of anyone who happens to wear AprilTags on their ankles. It runs on a Raspberry Pi with Simulink for both control modeling and kinematics.



Kinematics Overview:

Christopher's arm contains 3 joints. The joint types are revolute, prismatic, revolute. Joint 1 is a revolute joint that actuates the circular base of the arm, allowing it a full 360 degrees of motion. Joint 2 is a prismatic joint, performing one diagonal linear extension motion of the arm. Joint 3 is a revolute joint positioned at the wrist of the prismatic arm, and gives the end-effector pitch control (with the intent to preempt variation in leg sizes and positions). Christopher will identify a target via an AprilTag in its 'search' mode, drive up to it, rotate around its center to directly face the leg, extend its arm and then clamp the target's ankle/calf.

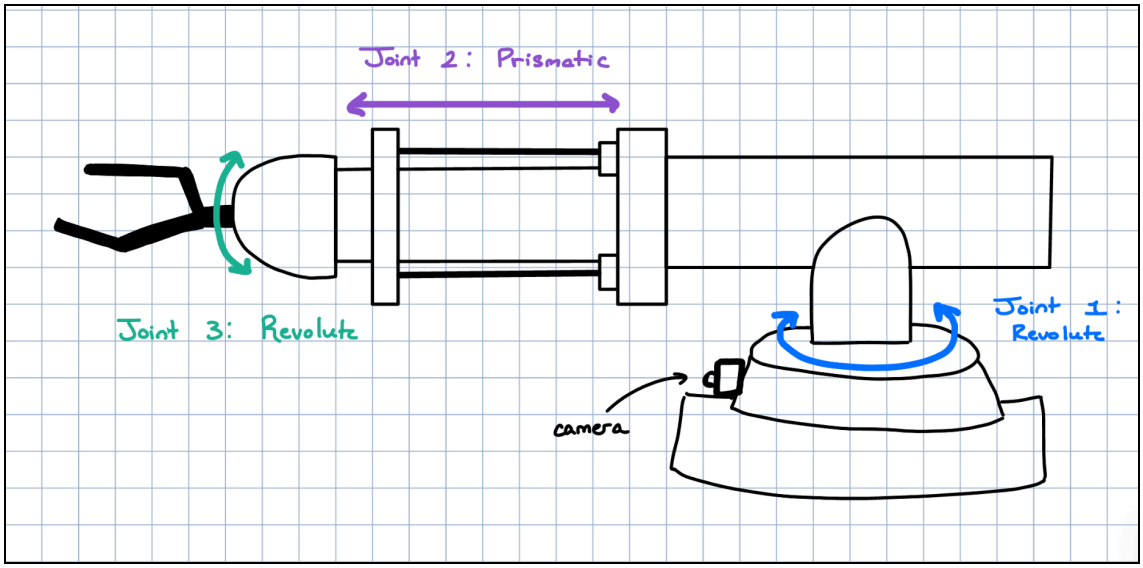


Figure 1: Christopher

Definition of Coordinate Frames and Denavit-Hartenberg Parameters:
DH Frame Table

Link *i*	α_i	d_i	a_i	θ_i
1	$\pi/2$	67.281	0	q_1
2	$-\pi/2$	$375.781 + q_2$	0	$\pi/2$
3	$-\pi/2$	0	78.2975	q_3

Where $q = [\theta_1\ d_2\ \theta_3]$

Table 1: DH Table for Christopher

Limits:
q[0]: $-\pi$ to π
q[1]: 0-187.908mm [the full extension of the arm]
q[2]: $-\pi$ to π

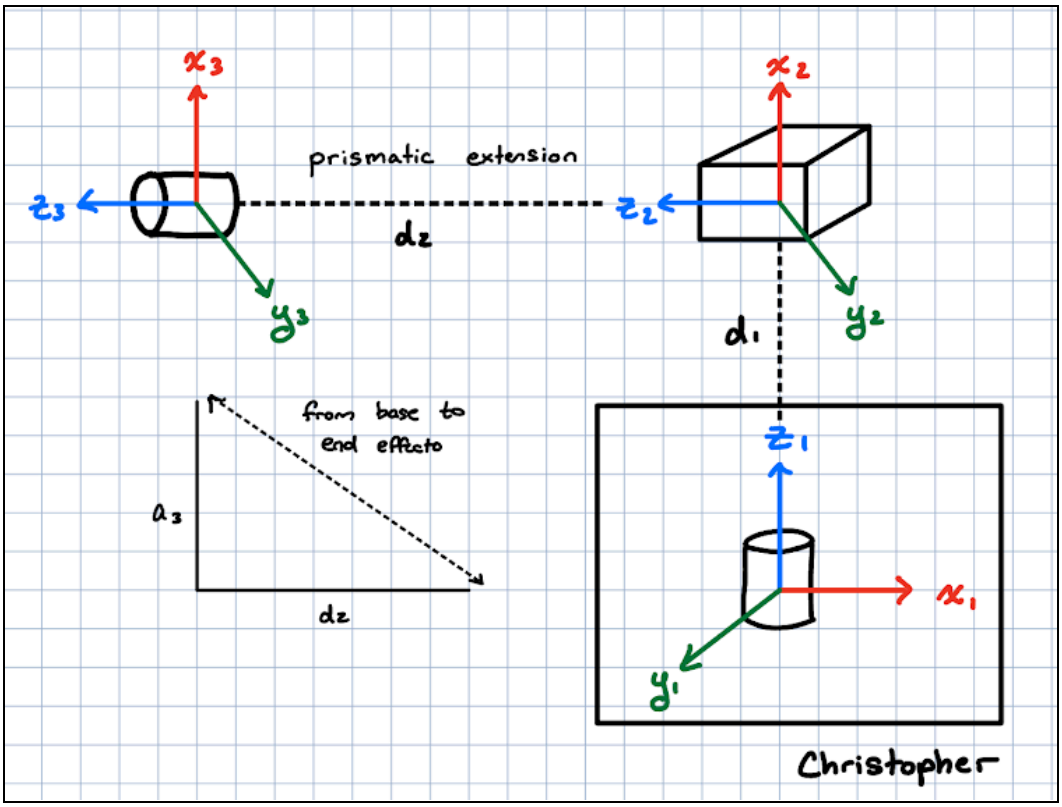


Figure 2: Frames for Christopher

Forward Kinematics Derivation:

Forward kinematics describes the process of changing the representation of a manipulator position from a joint space description into a Cartesian space description.

The equation is described by $\{\mathbf{y} = \boldsymbol{k}(\boldsymbol{q})\}$ where $\boldsymbol{q}$ represents the joint vector input of size $\mathbf{n}$ (where $\mathbf{n}$ is the number of degrees of freedom), $\boldsymbol{k}$ is defined by the transformation function derived from our Denavit-Hartenberg Convention, and $\mathbf{y}$ represents the final pose (position [x, y, z] and orientation [rotation matrix $\mathbf{R}$]) of the end-effector in Cartesian space.

Given the 3 joints defined (1 prismatic and 2 revolute) in the system of our robot, there are 5 links that correspond.

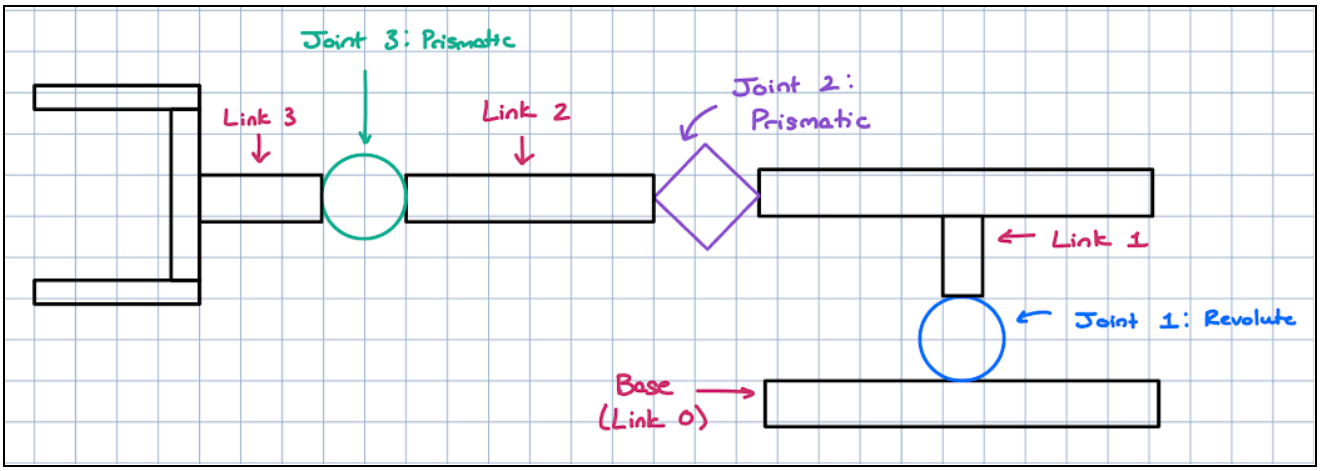


Figure 3: Link and Joint Mapping

$$A_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_i \\ \sin\theta_i \cos\alpha_i & \cos\theta_i \cos\alpha_i & -\sin\alpha_i & -\sin\alpha_i d_i \\ \sin\theta_i \sin\alpha_i & \cos\theta_i \sin\alpha_i & \cos\alpha_i & \cos\alpha_i d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Figure 4: Classical Denavit-Hartenberg Convention for Forward Kinematics

To specifically compute the forward kinematics, the homogeneous transformation matrices of each link is required (A_1 - A_4) and they are multiplied together to compute the final transformation matrix (T).

For this example, we computed the forward kinematics of the original DH table orientation for the end-effector:

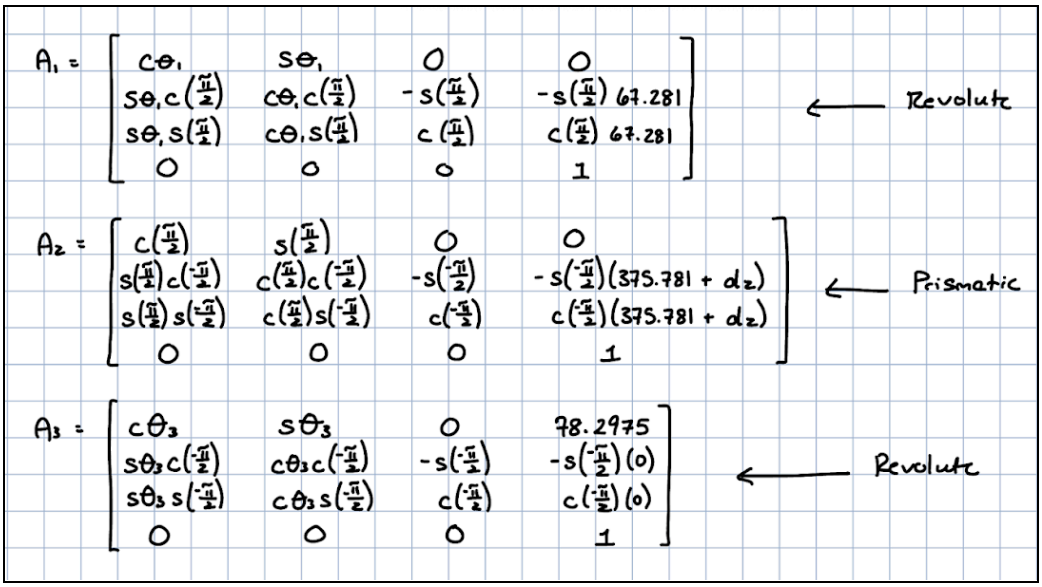


Figure 5: Evaluation of Homogeneous Transformation Matrices

```
Our computed transformation matrix for the forward kinematics of our robot is:
    0.0000    1.0000   -0.0000    0.0000
    0.0000    0.0000    1.0000 -375.7810
    1.0000   -0.0000   -0.0000  145.5785
         0         0         0    1.0000

Checking against fkine of the Robotics Toolbox:

T_toolbox =
         0         1         0         0
         0         0         1    -375.8
         1         0         0     145.6
         0         0         0         1

Difference between computed and toolbox:
   1.0e-31 *

    0.1233         0         0         0
   -0.1233         0         0         0
         0         0         0         0
         0         0         0         0
```

Figure 6: Computed Transformation Matrix for Forward Kinematics and Comparison

Workspace Visualization and Representative Robot Configurations:

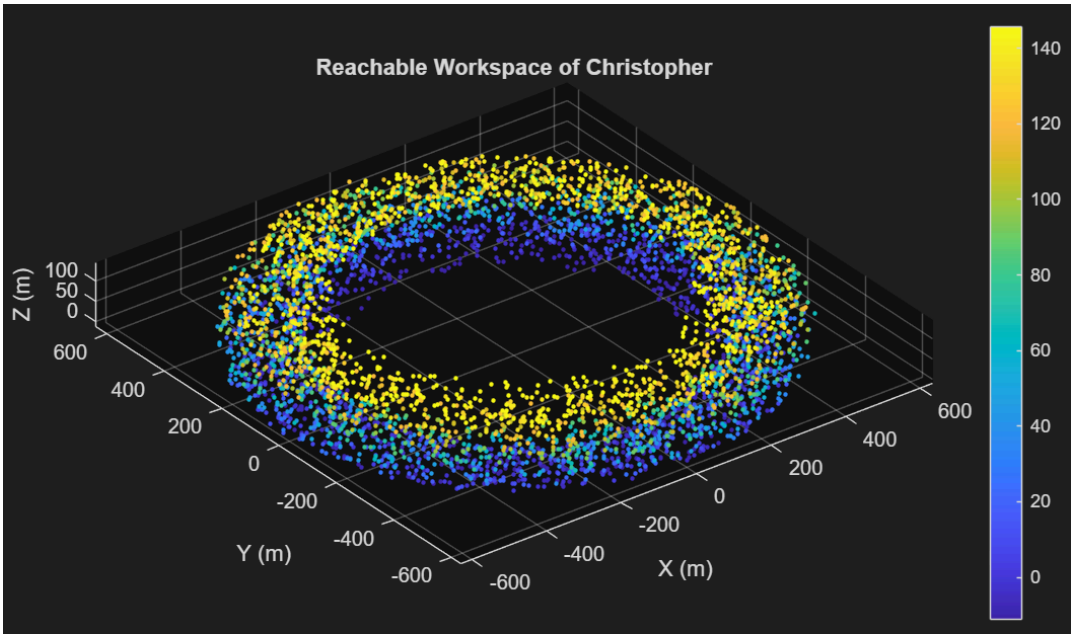
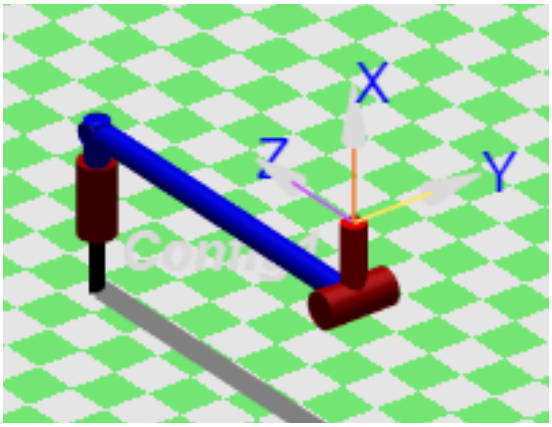
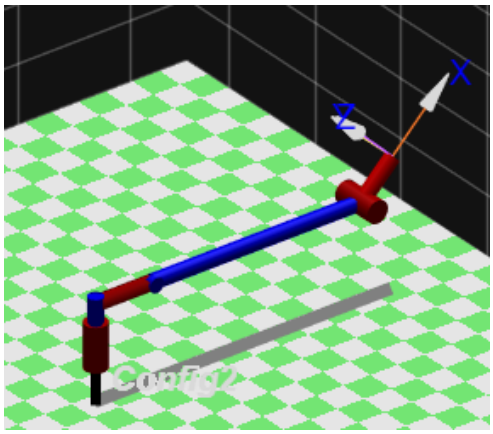


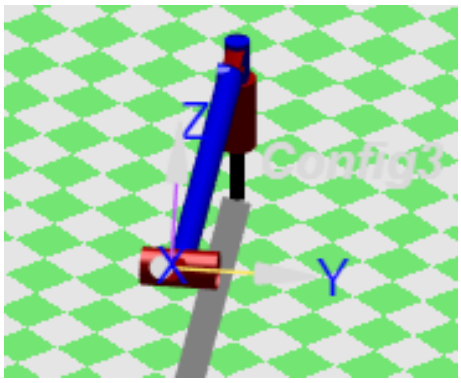
Figure 7: Reachable workspace of Christopher



q = [0 0 0]



q = [π/2 100 -π/4]



q = [-π/4 50 -π/3]

Figure 8: Different configurations of Christopher

*note: workspace dimensions are set as: ws = [-700 700 -700 700 -100 400] so that the full robot (including extensions) can be captured → hence why the entire plane is not captured since it is too large

Description of Simulation Implemented in MATLAB:
To implement our simulation, we first allocated and assigned links (L(1-3)) based on the conventions stated in our Denavit-Hartenberg table, in addition to assigning a 'q' vector. Simulating the robot produced the configurations seen above.

Following this, to derive a computation of our forward kinematics through matrix multiplication, the 'linkTransform' function was created and set with specific parameters for both prismatic and revolute joints. This computation was then checked against the '.fkine' method in Peter Corke's Robotic Toolbox to assure that it was accurate.

Inverse Kinematics and Jacobian Calculations:

Inverse Kinematics Solution:

For the inverse kinematics, 3 desired end effector positions were chosen. Two were within reach of the robot, as previously confirmed by forward kinematics. One was positioned out of limits to further test the method.

Position 1 (within reach) =

0.4330	0.8660	-0.2500	271.8
-0.7500	0.5000	0.4330	-470.8
0.5000	0	0.8660	106.4
0	0	0	1

Position 2 (within reach) =

-0.7071	0.7071	0	-427.1
-0.7071	-0.7071	0	-427.1
0	0	1	67.28
0	0	0	1

Position 3 (out of reach) =

1	0	0	1000
0	1	0	1000
0	0	1	1000
0	0	0	1

The resulting inverse kinematics calculations used the Peter Corke Robotics Toolbox to evaluate the desired joint angles and extensions to achieve each position (Appendix A). The following valid q vectors were found for positions 1 and 2. Inverse kinematics failed to converge for position 3, as expected.

Position 1:

```
True Solution according to FK:  q1=30.00 deg | q2=100.0000 mm | q3=-60.00 deg
Inverse Kinematics Solution:   q1=30.00 deg | q2=100.0045 mm | q3=-59.99 deg
Position error: 1.0880e-02 mm
Result: SUCCESS
```

Position 2:

```
True Solution according to FK:  q1=-45.00 deg | q2=150.0000 mm | q3=-90.00 deg
Inverse Kinematics Solution:   q1=-45.00 deg | q2=150.0138 mm | q3=-90.00 deg
Position error: 1.4257e-02 mm
Result: SUCCESS
```

Position 3:

```
Inverse Kinematics Solution:   q1=134.82 deg | q2=187.9080 mm | q3=-44.04 deg
Position error: 2.0508e+03 mm
```

`Result: FAILED`

In addition, we also calculated the inverse kinematics numerically. We used forward kinematics to derive the final position vector, which was used for inverse kinematics. For q3, we solved it as it was isolated, then we used the radial distance from the z-axis in order to solve q1 and q2. The full calculations can be found in Appendix C.

Task-space Trajectory or Waypoint Planning:
For the combined task-space trajectory and waypoint planning, 2 arbitrary waypoints (in the range of the figure) were chosen to demonstrate the entire trajectory of our robot (Appendix B).

Waypoint 1 = `[0.5 0.3 0.25]`
Waypoint 2 = `[0.5 -0.3 0.25]`

From these two points, there are three trajectories that were plotted:

1. The cartesian line [blue] demonstrates our prismatic joint (in the xy-plane)
2. The joint space 1 arch [red] demonstrates our first revolute joint (in the xy-plane)
3. The joint space 2 arch [green] demonstrates our second revolute joint (in the yz-plane)

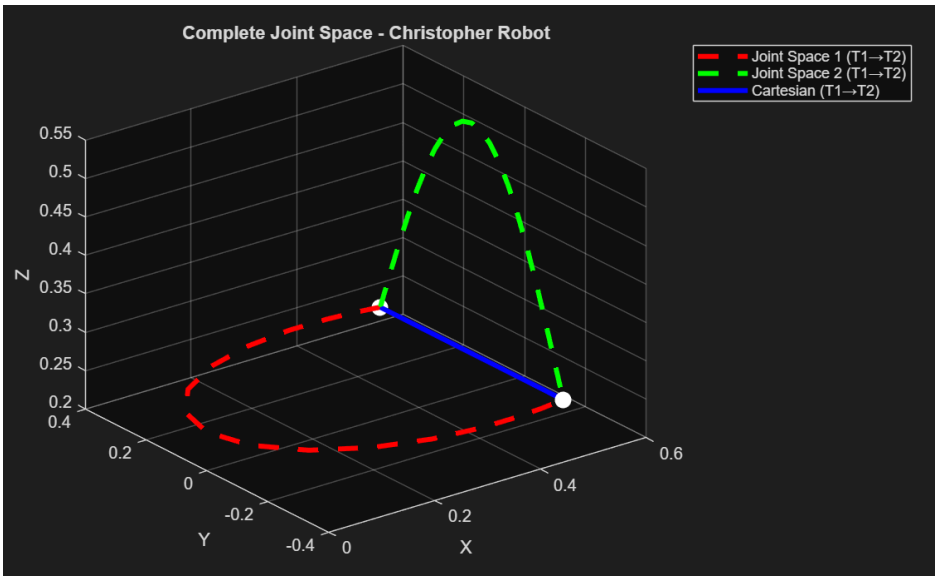
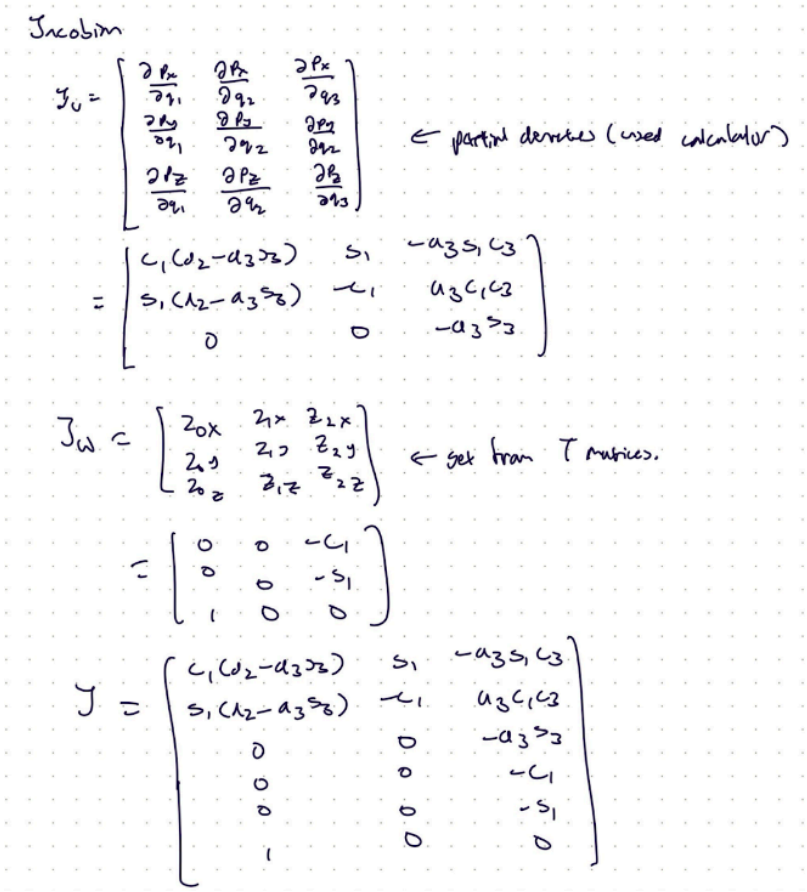


Figure 9: Task-space trajectory and waypoint planning graphed in MATLab

To best visualize the trajectory of our robot on one graph, we utilized the joint spaces in such a way that centered their path along the cartesian line. However, it is important to note that the joint space 2 arch [green] is motion that is solely reserved for the end-effector (it does not affect the link 1 of the arm).

Jacobian Derivation or Numerical Computation:

The Jacobian is a fundamental tool utilized in robotics that specifically relates joint velocities to Cartesian end-effector linear and angular velocities. We derived the Geometric Jacobian, which involves doing partial derivatives of the position vectors in order to get linear velocity. Then, we did the same for angular velocity, with z0, z1, and z2. z0 is the base so it goes straight up. z1 is about the prismatic, so there is no velocity there. Finally, z2 is derived from the transformation from multiplying T1 and T2 during forward kinematics.



Refer to Appendix C to see where the values are derived from.

We also used the Peter Corke Robotics Toolbox to compute the Jacobian, define an example desired end effector motion (dx = [2.0; 0; 1.0; 0; 0; 0], small motion in the X and Z directions), then compute the damped pseudoinverse, the joint update, and add the joint update to the original configuration (Appendix A).

```
Original position:
    271.7943
   -470.7616
    106.4298

New position:
    273.7920
   -470.7512
    107.4252
```

```
Position change:
      1.9977
      0.0103
      0.9955
```

Joint limits and kinematic singularities

Each join in the arm is held to physical constraints which restrict the reachable workspace of the robot and affect feasible trajectories. Our R-P-R model has defined the joint limits as:

Joint 1: -pi < q < pi,

Joint 2: 0 < q < 642mm,

Joint 3: -pi < q < pi.

The joint limits are the cause of several important restrictions, namely that (1) the robot cannot reach all positions or orientations in the space around it, (2) the prismatic joint is a direct limit to how far our arm can extend horizontally while stationary, and (3) the revolute joints restrict the angular reach and limit the orientations that the arm is capable of reaching. Even when initial and final end effector poses are valid, intermediate configurations along a trajectory may violate these bounds, requiring careful verification when using inverse kinematics. If one is not careful when calculating joint angles during inverse kinematics, they could find that the joint will not move to the required position. In addition, kinematic singularities occur when the Jacobian matrix loses rank, resulting in a loss of one or more degrees of freedom in task space. At such configurations, the robot cannot generate motion in certain directions and small cartesian velocities may require excessively large joint velocities, leading to instability. For this robot, singularities can arise when the prismatic joint approaches zero extension, like q2 = 0 or 642mm, or when the links align in configurations such as q3 = pi or $-\pi$, causing the Jacobian columns to become linearly dependent. These conditions were analyzed numerically using tools from the Peter Corke Robotics Toolbox by evaluating the rank of the Jacobian at different configurations. Understanding and avoiding joint limit violations and singular configurations is essential for ensuring smooth, stable, and physically feasible robot motion.

Sensor Overview

I. Pi Camera

The Pi Camera is calibrated using a chessboard calibration procedure to compute the camera intrinsic matrix and distortion coefficients. These parameters are used to correct image distortion and enable accurate geometric measurements. AprilTags are detected using a Python script running on the Raspberry Pi. The script computes the tag's relative position (x,y,z) using a Perspective-n-Point (PnP) algorithm and sends the data via UDP.

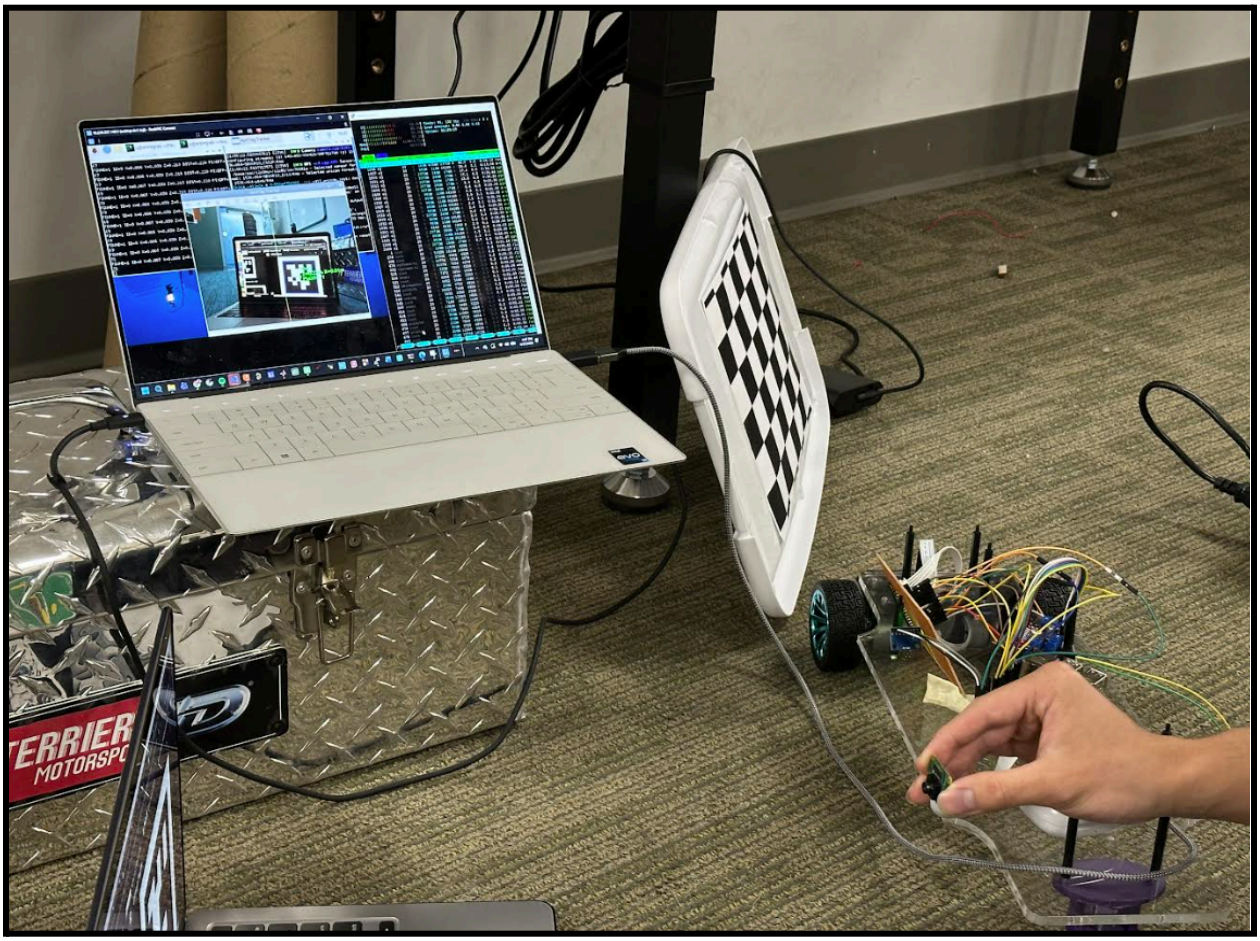


Figure 10: Calibrating Pi Camera Intrinsics

The purpose of the perception block, where pi cameras is integrated in Simulink, is to convert raw camera data into meaningful spatial information that can be used for control. Using prior camera calibration (intrinsic matrix and distortion coefficients), the system corrects image distortion and accurately identifies the pixel coordinates of the tag corners. This coordinate information is sent to Simulink via UDP and parsed in the **Perception** block, which outputs structured signals such as detection status, position, and pixel offset. This process is necessary because the robot must convert visual information into real-world coordinates in order to

determine how far and in what direction to move. Without this transformation, the system would only have pixel data, which is insufficient for motion planning and closed-loop control.

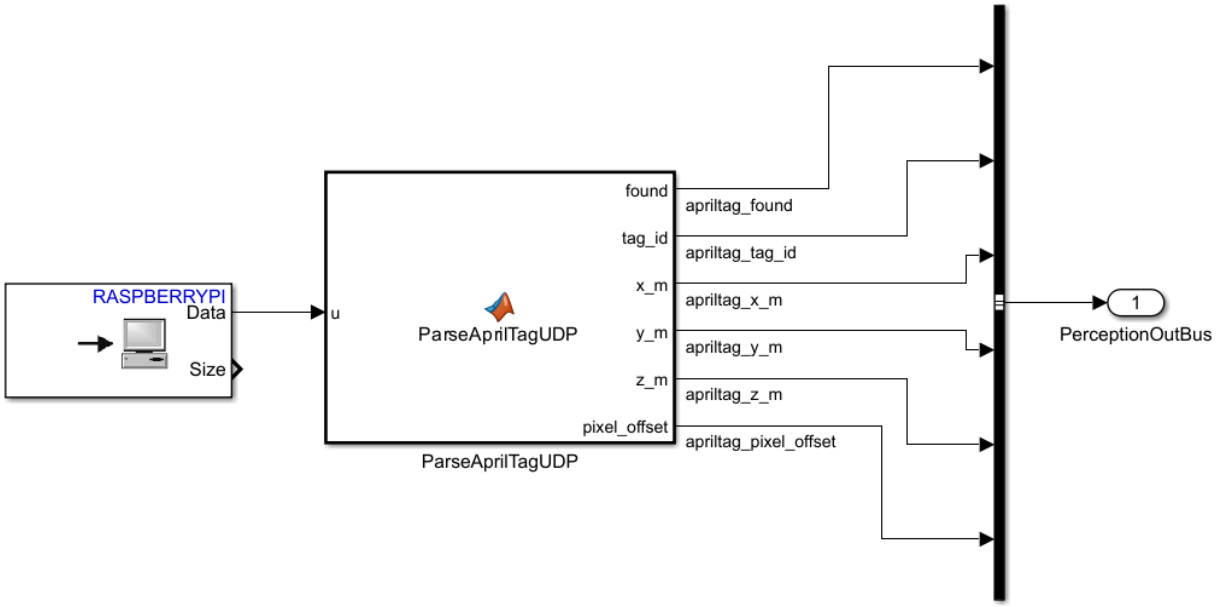


Figure 11: Pi Camera data input into Simulink via UDP

II. DC Encoders

The DC motor encoders are incremental encoders that generate pulses as the wheel rotates. By counting these pulses (ticks) over time, the system computes wheel angular velocity and total distance traveled. Using both left and right wheel encoders, the robot estimates its motion (odometry): forward movement comes from the average of the wheel speeds, and turning comes from the difference between them.

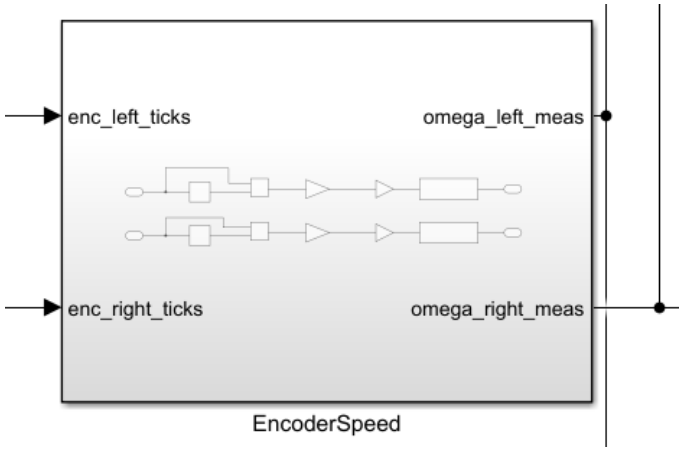


Figure 12: Encoder Speed Subsystem

This information is continuously fed into the control system to provide feedback for the PI controllers and to estimate the robot's position and orientation. Odometry is essential for autonomy because it allows the robot to know how it is moving even without external sensors. It also supports motion planning by enabling the system to track where it has gone and adjust its commands in real time based on actual movement.

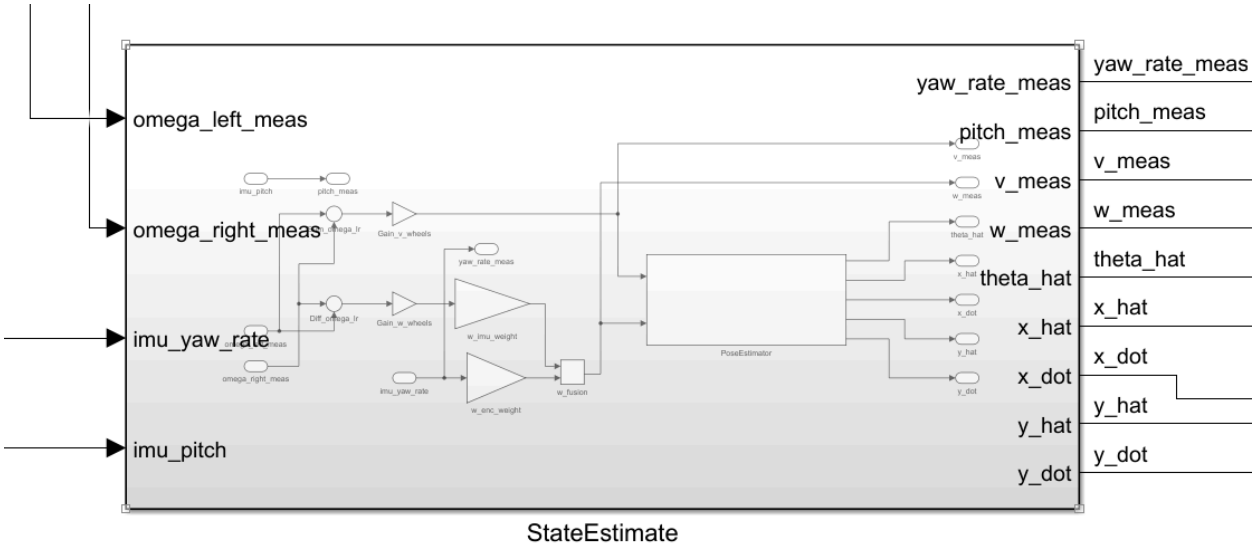


Figure 13: StateEstimate System

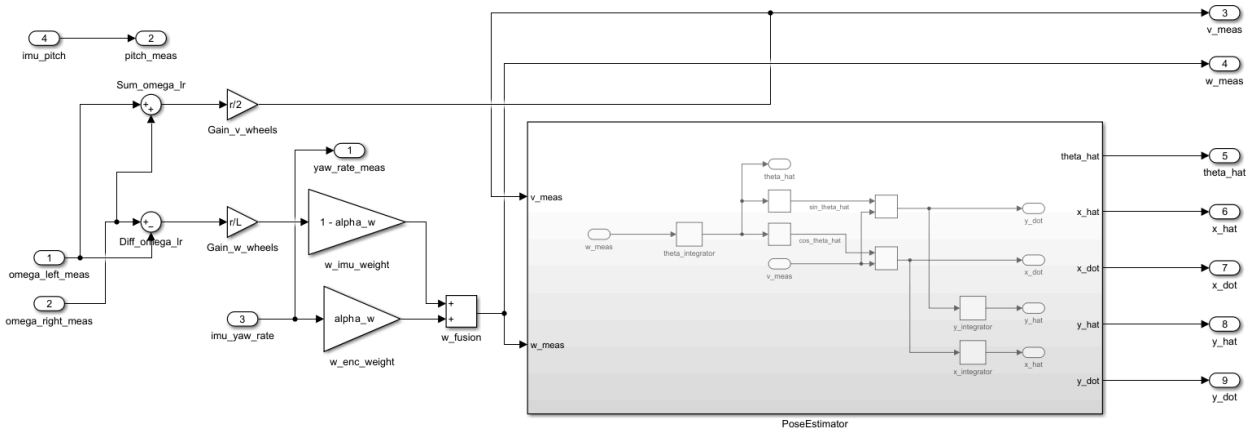


Figure 14: StateEstimate Subsystem & PoseEstimate Subsystem

Odometry uses incremental encoder ticks to estimate how far each wheel has traveled. Each full wheel revolution corresponds to a known number of ticks, so by counting ticks we can compute distance using the wheel circumference. The robot's forward motion is found from the average of the left and right wheel distances, while rotation is determined from their difference. Starting

from an initial position of (0,0) at power-on, these values are integrated over time to estimate the robot's pose (x, y, and heading). This provides a continuous estimate of position for control and motion planning without needing external references.

III. IMUs

The IMUs are used to provide orientation feedback in parts of the system where direct position sensing is unavailable or unreliable. One IMU is mounted on the robot base to measure yaw rate, which helps stabilize heading and improves autonomous navigation by correcting drift that cannot be captured perfectly by wheel encoders alone. A second IMU is mounted on the stepper-driven joint to track yaw rotation, allowing more accurate control of the base rotation. A third IMU is placed on the femur/end-effector to estimate its orientation, since the servo motors do not provide true position feedback.

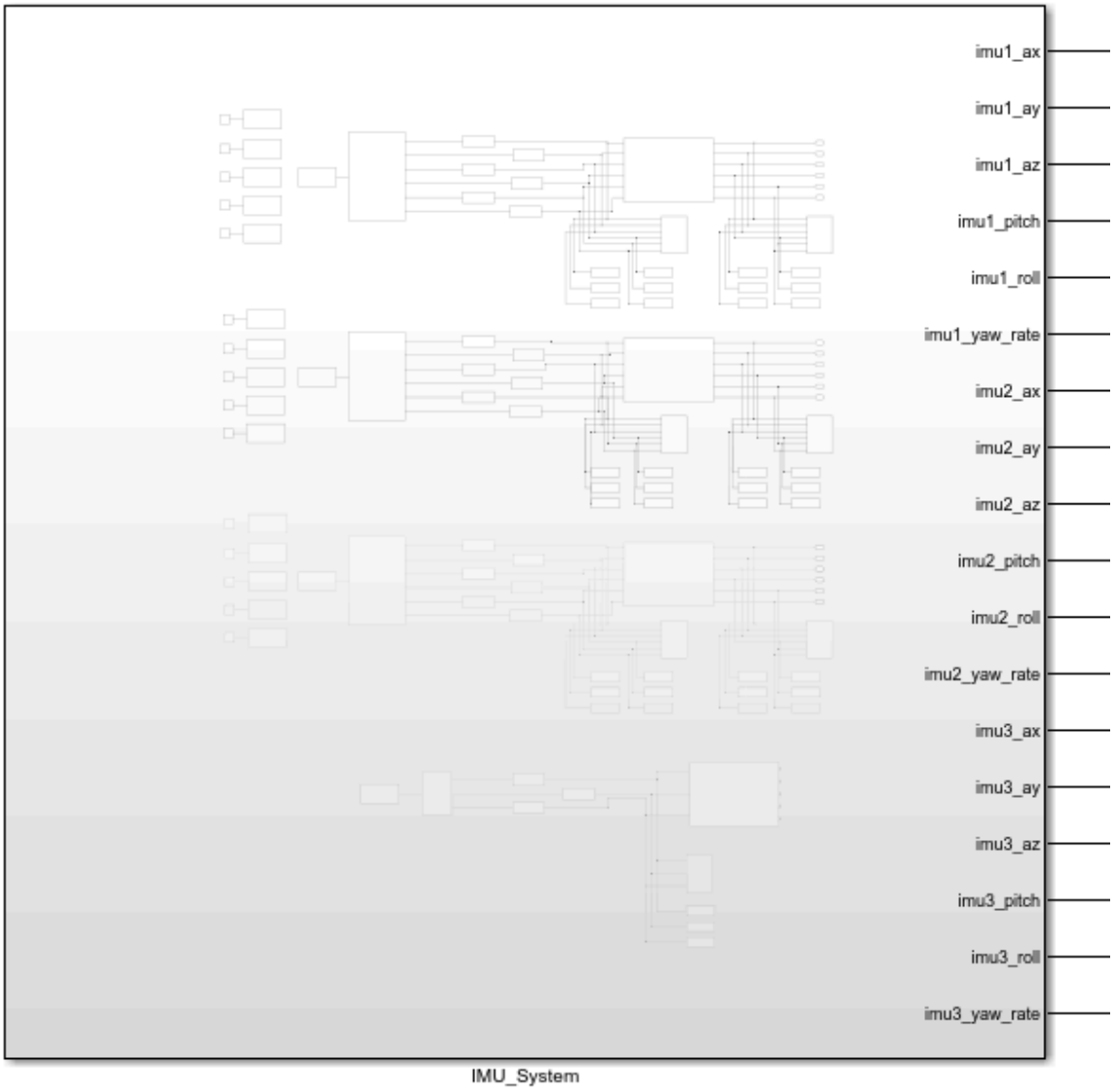


Figure 15: IMU Subsystem w. all 3 IMUs embedded parsing via I2C + Kalman Filter

The IMUs are interfaced using I2C, where the Raspberry Pi communicates with each sensor by writing to and reading from specific register addresses defined in the MPU6050 datasheet. Because only a limited number of I2C addresses are available, one IMU's address pin is pulled low to ensure unique addressing on the shared bus. The system initializes the IMU by writing to configuration registers, then reads consecutive data registers in a single transaction to efficiently retrieve raw acceleration and gyro measurements. These raw byte values are parsed in software using datasheet scaling formulas to convert them into physical units such as m/s^2 and rad/s. A Kalman filter is then applied to combine noisy measurements and model predictions, producing a smoother and more accurate estimate of orientation and angular velocity for use in closed-loop

control.

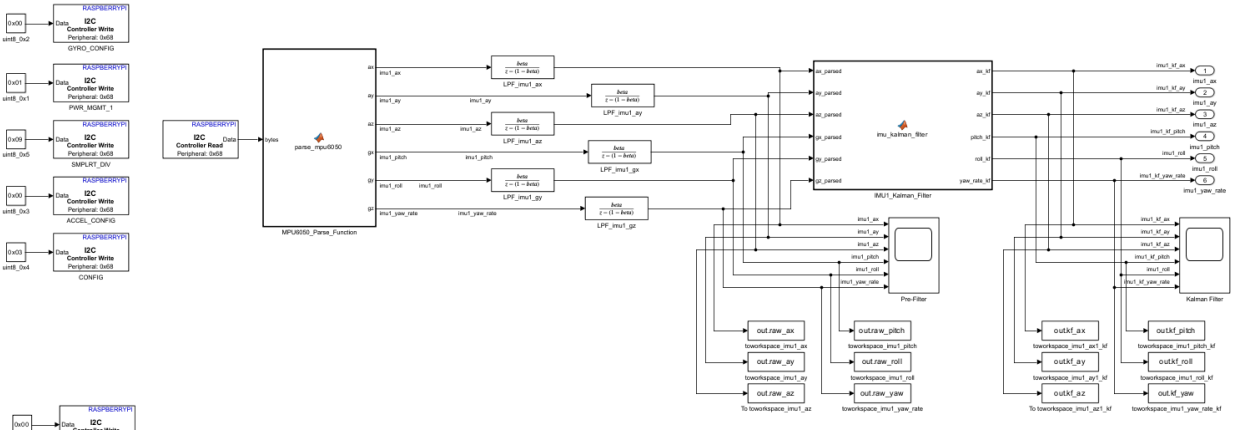


Figure 16: Embedded Bytestream Parsing via I2C for MPU6050 via Simulink Raspberry Pi W/R blocks to access IMU config settings and data registers + Kalman Filter for noise reduction

At startup, the arm is zeroed to establish a reference frame, and the IMU is then used to track deviations from this reference. Because PWM-based servo control is affected by friction and mechanical uncertainty, the IMU feedback is used in a PI/PD control loop to correct for overshoot and undershoot, improving positioning accuracy.

VI. Ultrasonics

Ultrasonic sensors are placed on the sides of the robot to provide distance measurements for obstacle detection. These sensors are intended to detect nearby objects that may block the robot's path or obstruct the camera's view of the AprilTag. Although full obstacle avoidance was not fully implemented, the design allows the system to determine available free space on either side of the robot and choose a direction to maneuver around an obstruction. If the robot loses visual tracking of the AprilTag due to an obstacle, the ultrasonic sensors provide local environment awareness to guide motion. Using pose estimation from odometry, the robot can track how far it deviates while avoiding the obstacle and attempt to return to its original trajectory once the path is clear.

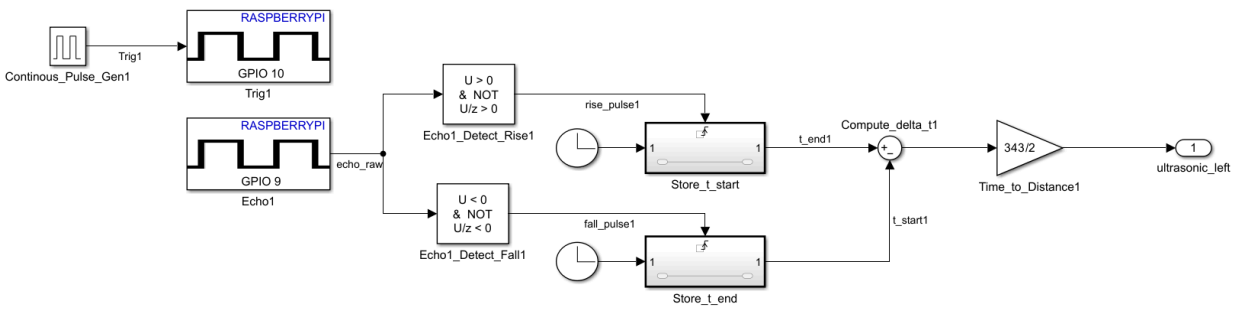


Figure 8: Ultrasonic sensor configuration

Motion Planning Overview

The motion planner is responsible for converting perception data into velocity commands that drive the robot toward the target. It operates using two primary modes, **Search** and **Chase**, which are selected by the FSM based on whether an AprilTag is detected.

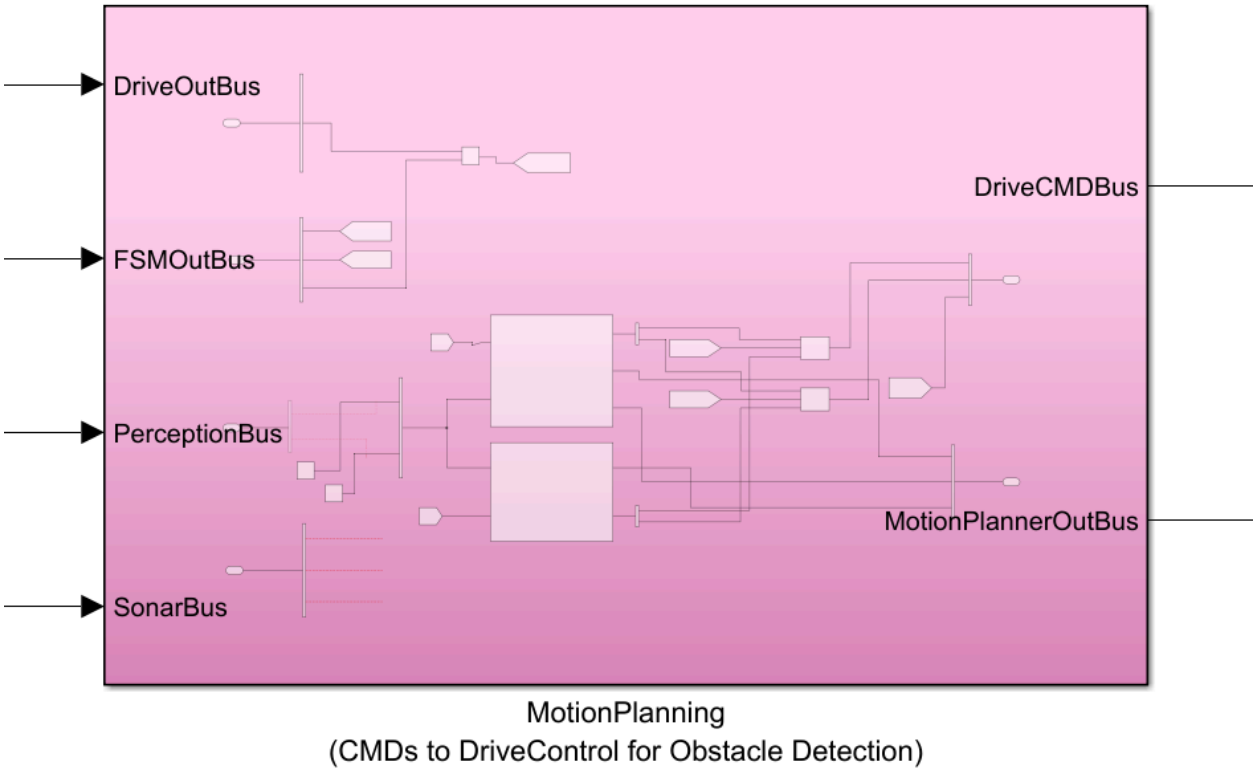


Figure 17: MotionPlanning Subsystem

In the Search subsystem, when no tag is found, the robot rotates in place by setting forward velocity to zero and applying a constant angular velocity, allowing the camera to scan the environment. Once a tag is detected, the system transitions to the Chase subsystem, where the robot begins moving toward the target. In this mode, forward velocity is increased to drive the robot closer, while angular velocity is adjusted using the pixel offset of the tag to keep it centered in the camera frame. The planner also includes logic to determine when the tag is "close enough," triggering a stop condition for precise interaction. If the tag is lost during motion, the system reduces forward velocity and reverts to a recovery behavior, preventing the robot from blindly continuing forward. These transitions and conditions are implemented using logical blocks that evaluate detection signals and distance thresholds. Overall, the motion planner provides a simple but effective strategy for autonomous navigation by combining perception feedback with reactive control decisions.

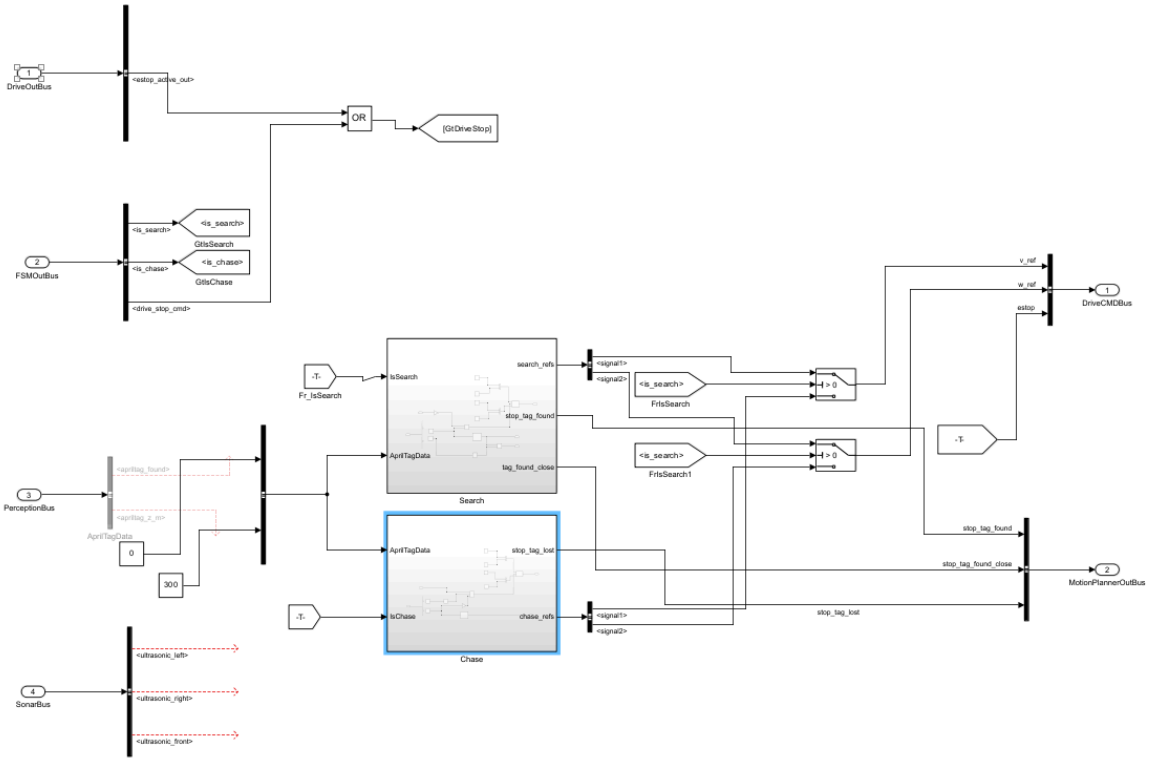


Figure 18: State Decision Making based on Perception and Sensor Inputs

FSM (Closed-Loop System) Overview

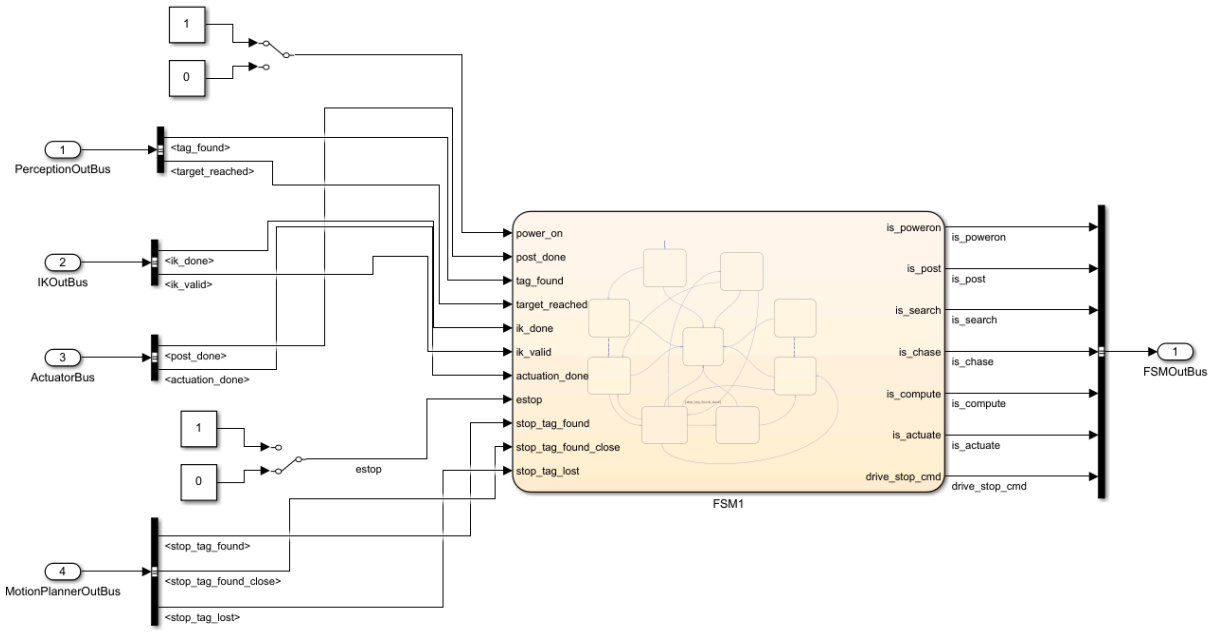


Figure 19: Finite State Machine Inputs with one-hot encoding outputs

The system uses a finite state machine (FSM) to coordinate perception, motion planning, and actuation in a closed-loop framework. Each state represents a distinct phase of operation, and transitions are triggered by sensor feedback and task completion signals.

At startup, the system begins in the **OFF** state, where all motion is disabled. Once power is enabled, the FSM transitions to the **POST** (Power-On Self-Test) state. In this phase, all actuators initialize and move to a known reference configuration (zero position), ensuring consistent starting conditions for the arm and drive system.

After initialization is complete, the system transitions to the **SEARCH** state. In this mode, the robot rotates in place by commanding a nonzero angular velocity and zero linear velocity. Because the robot is differential drive, this is achieved by driving one wheel while the other remains stationary, allowing the camera to scan the environment for an AprilTag. As soon as a tag is detected, the robot immediately stops, naturally aligning itself toward the tag due to its rotation behavior.

Upon detection, the FSM transitions to the **CHASE** state. In this mode, the robot drives toward the AprilTag using motion planner outputs (vref, wref). The linear velocity is positive to move forward, while angular velocity is adjusted to keep the tag centered in the camera frame. This creates a closed-loop visual servoing behavior that continuously corrects the robot's trajectory.

If the tag becomes occluded or lost (for example, due to an obstacle blocking the camera), the FSM transitions to a **RECOVERY/SEARCH** behavior. In this state, the robot stops forward motion and resumes scanning until the tag is reacquired. This ensures the system does not blindly continue moving without valid perception input.

When the robot reaches a predefined distance threshold to the tag (within reachable workspace), the FSM transitions to the **COMPUTE IK** state. Here, inverse kinematics are calculated to determine the required joint positions for the arm to reach the target. Once valid joint commands are generated, the system transitions to the **ACTUATE** state.

In the **ACTUATE** state, the arm executes the motion to interact with the target (e.g., grabbing the ankle). After the action is completed, the system transitions to the **DONE** state, where motion is halted.

An **ESTOP** state is available at any time, immediately disabling all motion if triggered.

Simulated Closed-Loop Performance

I. LQR Controller

The scope outputs were used to evaluate closed-loop performance and verify that each part of the system behaves as expected. The control signals **u_left** and **u_right** show a brief transient before settling, indicating that the PI controllers are stabilizing the wheel velocities effectively. The encoder tick signals increase linearly over time, confirming continuous wheel rotation and consistent velocity tracking. The measured forward velocity remains steady, while a small nonzero angular velocity is observed, which explains why the robot is not moving perfectly straight. This is reflected in the pose estimates, where **x_hat** and **y_hat** form sinusoidal-like curves, indicating circular motion due to constant forward and rotational velocity. The heading estimate **theta_hat** increases steadily over time, which is consistent with a constant turn rate. Additionally, the IMU yaw rate closely matches the encoder-based angular velocity, validating the sensing and state estimation. Overall, these results confirm that the system is operating as a stable closed-loop controller with physically consistent motion behavior.

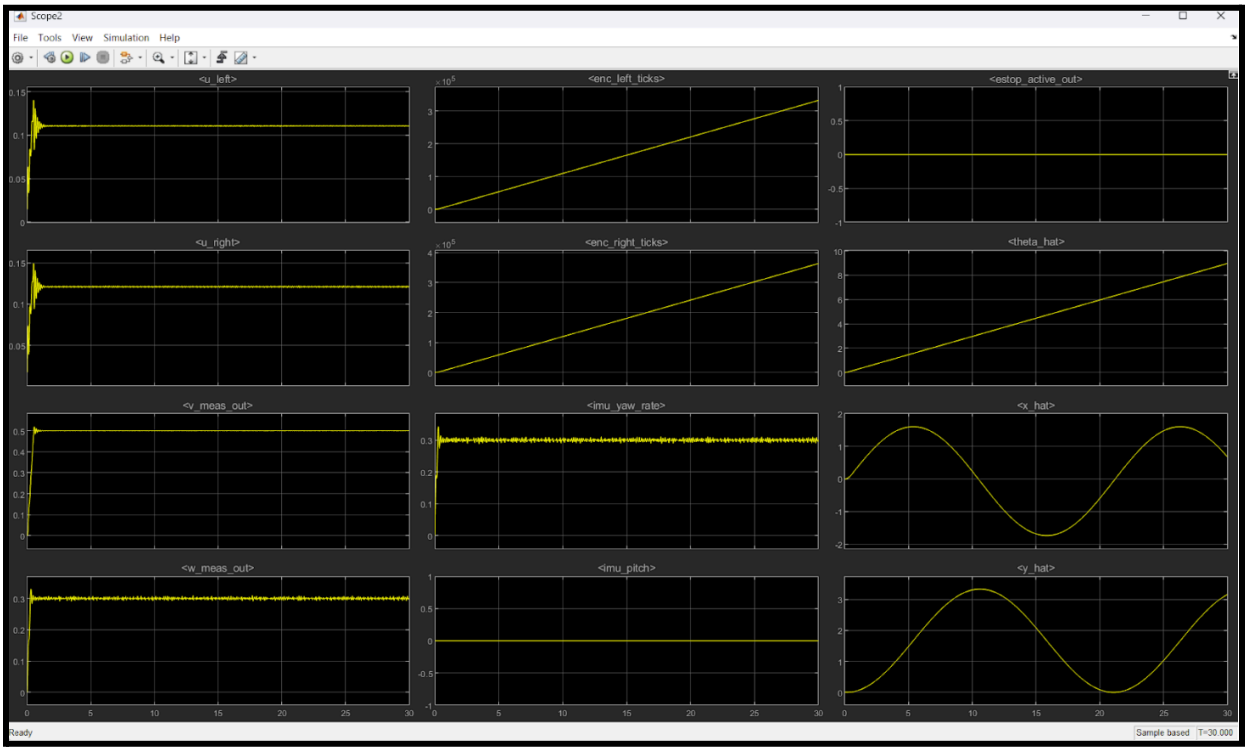


Figure 20: LQR Tuning/Scoping of encoder and imu values through simulated Plant

II. PID Controller

The LQR controller demonstrates fast and stable convergence with minimal overshoot due to its optimal state feedback design. The system settles quickly and maintains smooth control inputs, resulting in reduced oscillations and noise in the encoder and IMU signals. Overall, LQR provides more consistent and efficient closed-loop performance compared to classical tuning methods.

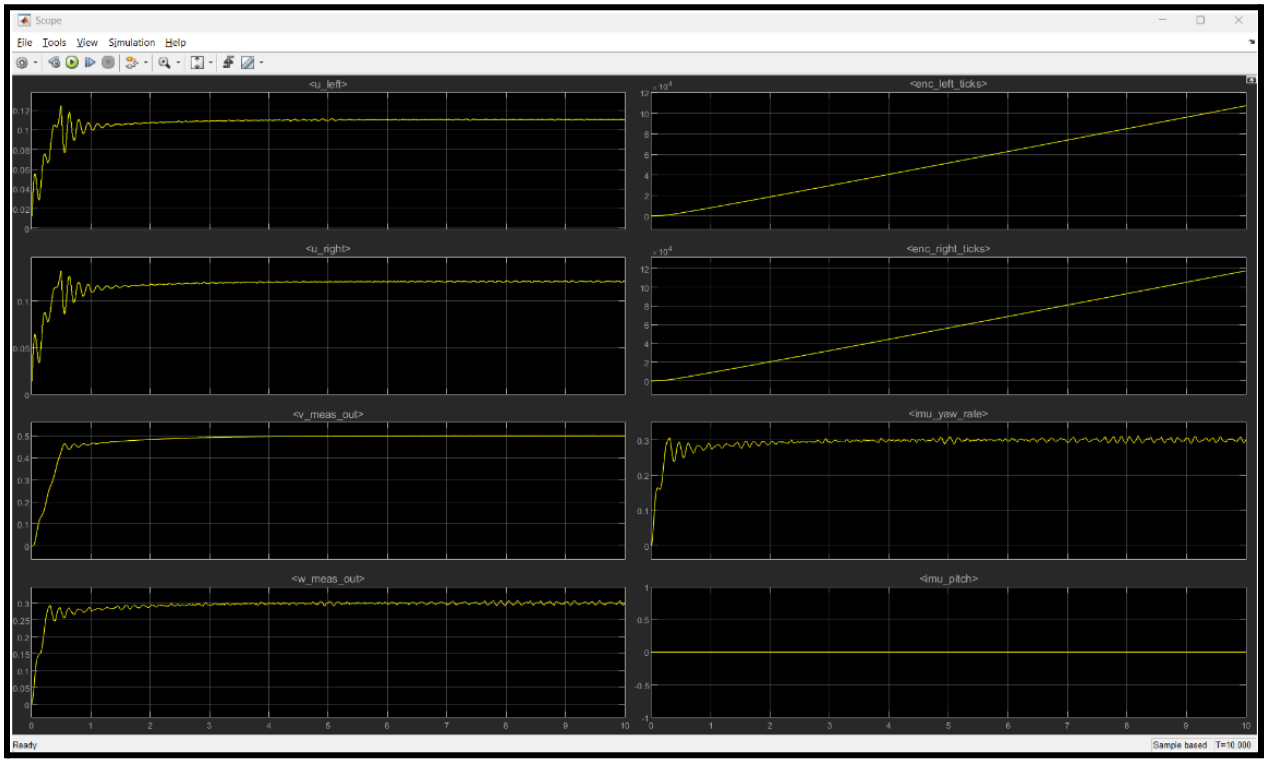


Figure 21: PID Tuning/Scoping of encoder and imu values through simulated Plant

The PID-controlled system exhibits a slower settling time with noticeable overshoot during the transient response. Encoder and IMU signals show more noise and small oscillations due to sensitivity to measurement error. While steady-state tracking is achieved, the system requires careful tuning to balance responsiveness and stability. Compared to LQR, PID control is more prone to oscillations and less optimal under dynamic conditions.

III. Plant

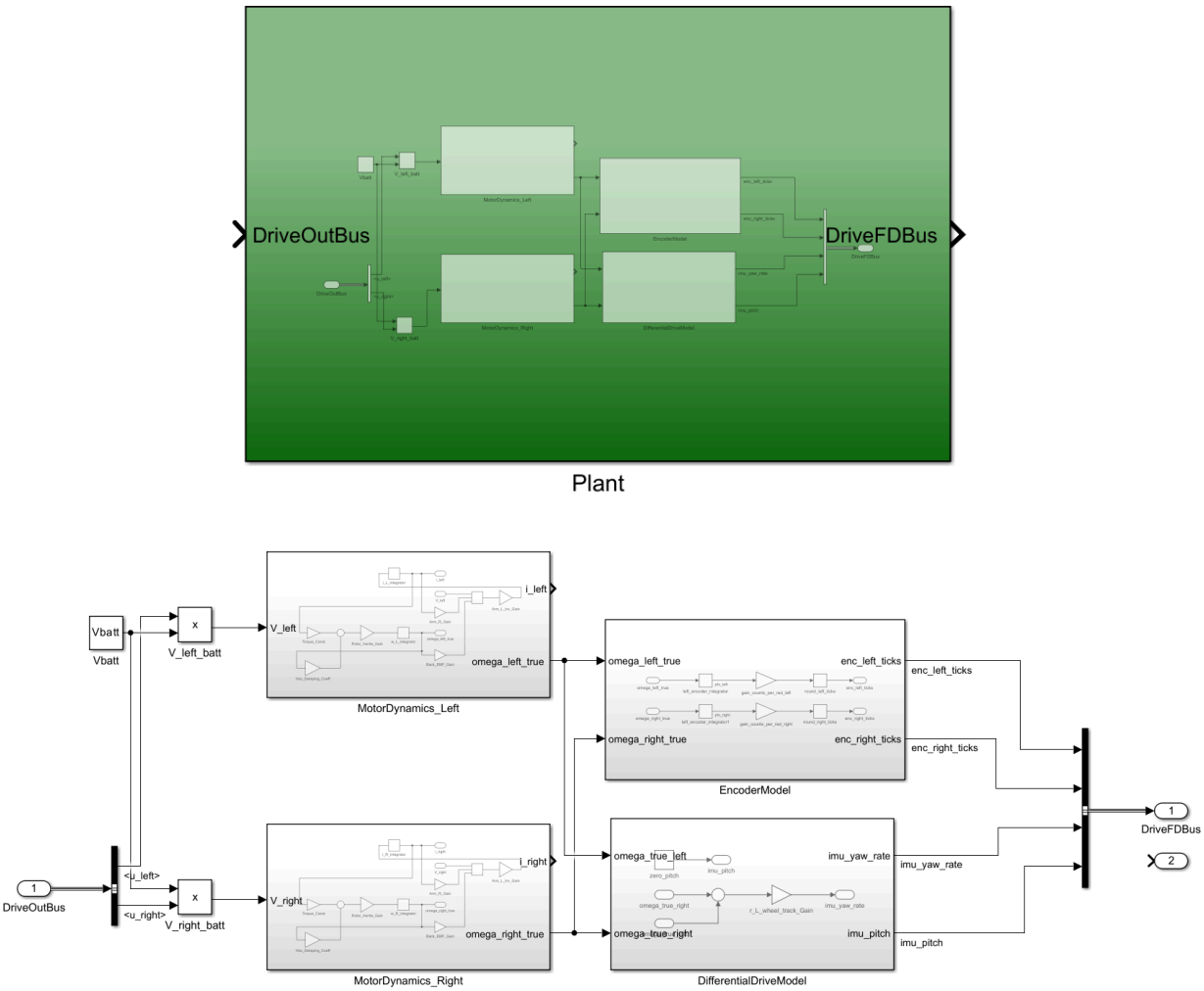


Figure 22: Plant for closed-loop simulation prior to hardware deployment

Before deploying on physical hardware, a plant model was developed in Simulink to simulate the robot's dynamics and enable controller design and testing. The model includes DC motor dynamics for each wheel, capturing electrical behavior (voltage, resistance, inductance) and mechanical effects such as inertia, torque generation, and back-EMF. These motor outputs produce wheel angular velocities, which are then passed into an encoder model to simulate tick counts, mimicking real sensor feedback. A differential drive model combines the left and right wheel velocities to estimate overall robot motion, including linear velocity, angular velocity, and orientation (yaw). This allows the system to replicate realistic robot behavior, including turning and curved trajectories. The plant model was essential for tuning controllers such as PID and LQR, validating closed-loop performance, and debugging the system before physical implementation. Overall, it provided a safe and controllable environment to ensure the full

sensing and control pipeline functioned correctly prior to hardware deployment.

Experimental Close-Loop Data

VI. IMU Pre-Filter vs Kalman Filter Data

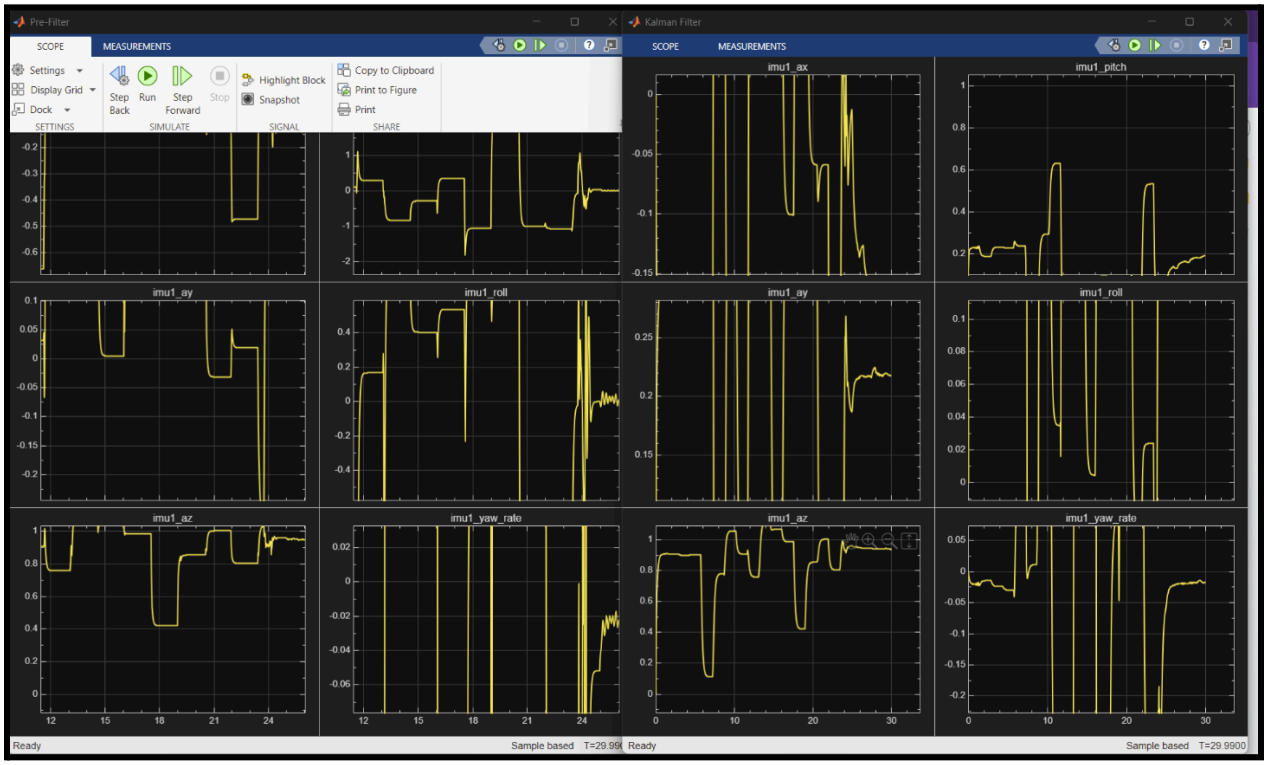


Figure 23: Real time IMU data on Simulink scopes with no filter vs Kalman Filter

The IMU plots compare raw/pre-filtered signals to the Kalman-filtered outputs, demonstrating the improvement in signal quality. In the pre-filtered data, the accelerometer and gyroscope signals exhibit sharp spikes, discontinuities, and high-frequency noise due to sensor imperfections and electrical disturbances. These fluctuations make the raw signals unreliable for direct use in control or state estimation. After applying the Kalman filter, the signals become significantly smoother and more continuous, while still preserving the overall motion trends. The Kalman filter works by combining the predicted state from a system model with noisy sensor measurements, weighting each based on their uncertainty to produce an optimal estimate. As a result, noise and sudden spikes are suppressed without introducing large delays. This is particularly evident in the yaw rate and orientation signals, where the filtered outputs track the general behavior of the raw data but with much less variability. Overall, the Kalman filter improves stability and accuracy, making the IMU data suitable for closed-loop control and state estimation.

V. Kart Real Time Encoder & IMU Deployment (view on Simlink)

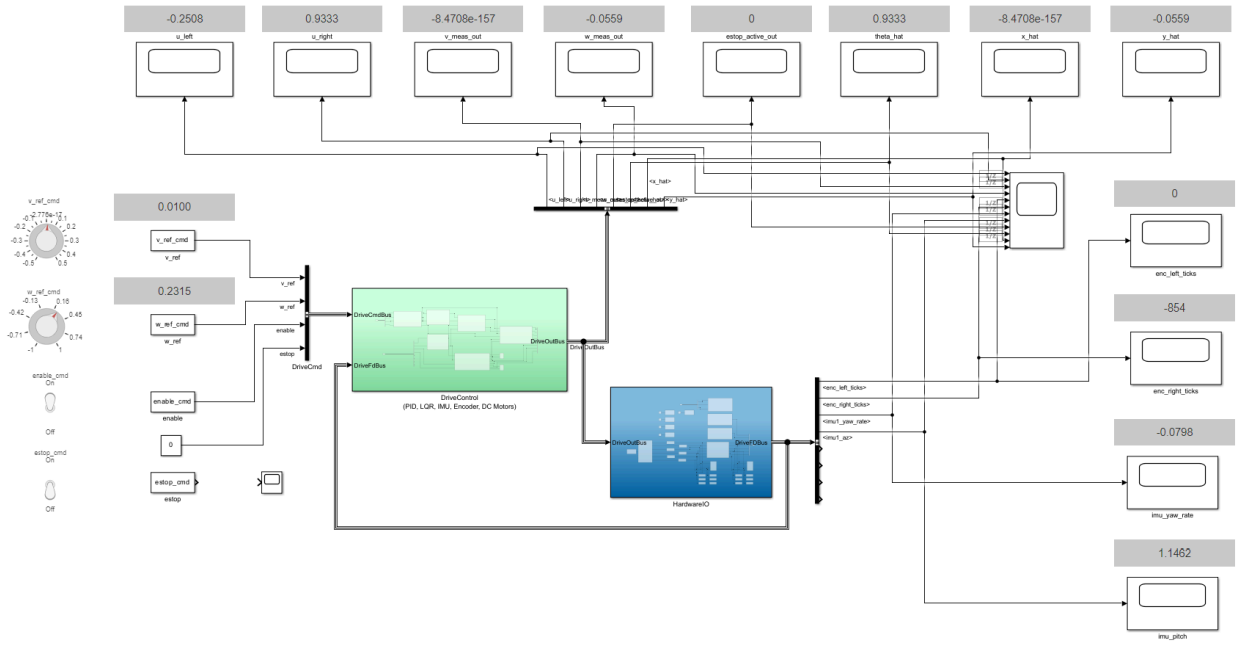


Figure 24: DC Wheel Encoders and IMU Scope Data during Real Time Deployment

The experimental results from physical deployment confirm that the closed-loop system operates as expected on hardware. The control inputs **u_left** and **u_right** remain bounded and respond appropriately to the commanded velocities, indicating stable actuator behavior. The measured forward velocity is near zero while a nonzero angular velocity is observed, showing that the robot is primarily rotating in place, which is consistent with the applied reference inputs. Encoder tick values change accordingly, verifying that the wheels are actively moving and providing valid feedback for odometry. The estimated heading **θ_hat** increases over time, while **x_hat** and **y_hat** remain small, confirming rotational motion about a fixed point rather than significant translation. Additionally, the IMU yaw rate closely aligns with the encoder-derived angular velocity, validating consistency between sensors. Overall, the hardware results demonstrate that sensing, estimation, and control are functioning together correctly in a real-world closed-loop system.

Final Comments

Derivations from Proposal:

Our main change from the proposal is the design of our robot joints. Instead of having a revolute joint on the end effector, we instead set it about the base. This allows us more flexibility in terms of ankle grabbing. Additionally regarding the physical chassis, we were initially planning to have the prismatic part of the arm at a 45 degree angle upwards to reach higher on the leg. However, after looking over the design constraints caused by this as well as seeing how high this put our end-effector, we were forced to redesign it to stick out straight and be parallel with the ground.

We also didn't implement the ultrasonic modules as expected, as ultrasonics introduced extra complexity that we didn't have time to integrate. For us, ankle detection via camera was paramount. With regards to other robotics languages like OpenCV ROS2 — at first we attempted to get OpenCV to work but quickly found that the processing of OpenCV paired with running a control framework would be too computational intensive for the Raspberry Pi, while ROS2 was redundant when used alongside Simulink. We opted for AprilTags for ease of detection and more reliable depth sensing, and to continue fully scripting our model in simulink.

Modeling, Sensing, and Control:

For modeling, we assume rigid body construction and ideal joints. We also assume that our assembly doesn't introduce friction along the prismatic and revolute joints, where we use ball bearings to lighten the load and reduce friction.

Key Equations:

Controls:

PID:

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t},$$

LQR:

$$J(u) = \int_0^\infty (x^T Q x + u^T R u + 2x^T N u) dt$$

Estimation:

Kalman Filter:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k \mathbf{u}_k$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^\mathsf{T} + \mathbf{Q}_k$$

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1}$$

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^\mathsf{T} \mathbf{S}_k^{-1}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k$$
$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\, \mathbf{P}_{k|k-1}$$

Our model can be described in Simulink (Figure 11), in which different functions are broken down into blocks.

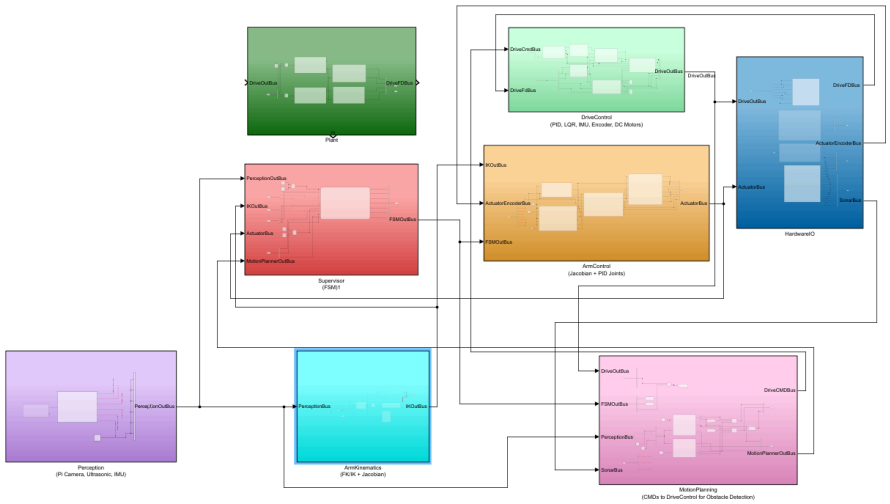


Figure 25. Simulink Model Overview

There are 7 blocks that make up our model.
Plant is used for simulation and testing.

Supervisor (FSM)	– Holds and translates our FSM states.
Perception	– Interfaces with Pi Camera and performs processing to get AprilTag data.
Motion Planning	– Determines driving behavior of the car like obstacle avoidance.
Drive Control	– Implements PI tuning in order to drive motors.
Hardware IO	– Manages interfacing with Pi GPIO and PWM pins to power motors.
Arm kinematics	– Calculates kinematics and Jacobian given AprilTag position.
Arm control	– Implements LDR and PID tuning for the robotic arm.

Our state logic is our FSM (Figure 12).

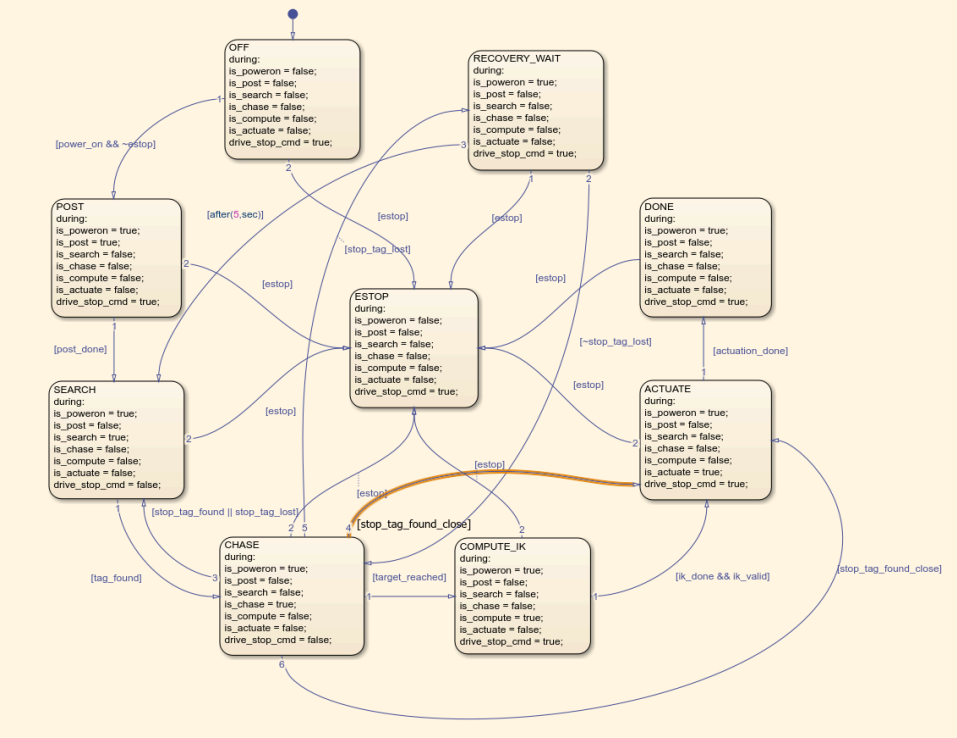


Figure 26

Results and Evaluations:

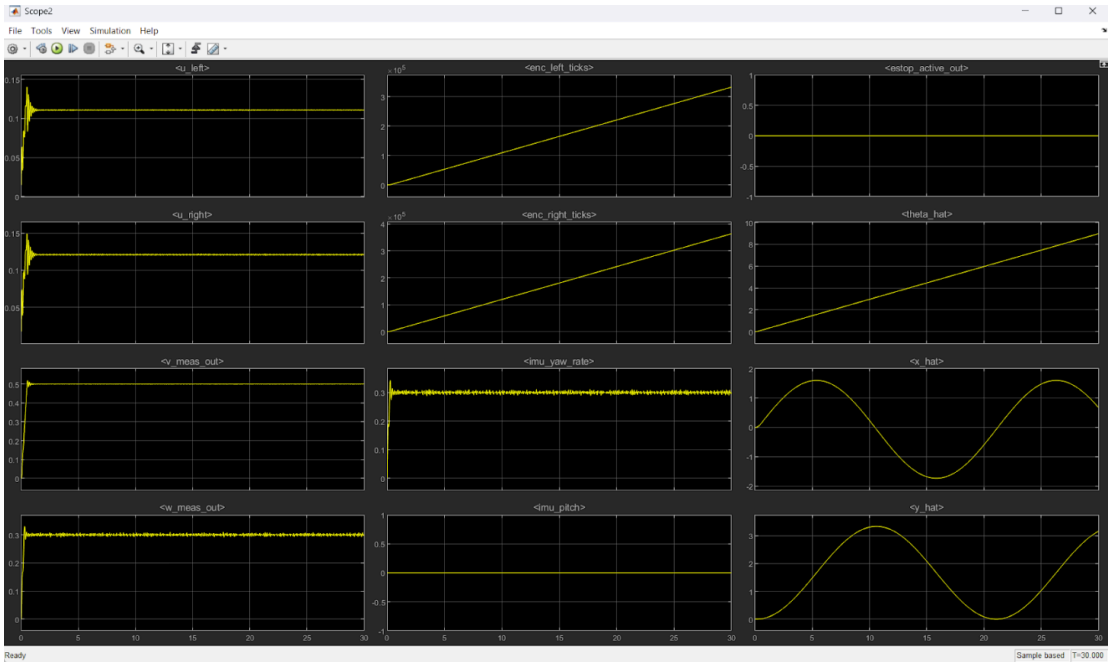
While we ultimately weren't able to integrate the core module of our simulink (perception) with the pi camera, both the model and the chassis-arm system were fully functional and able to be simulated independently. We were able to prove the desired functionality of our systems through:

- April tag detection
- Actuator test scripts
- Simulink simulation

Our pi camera did work independently and was ultimately able to detect AprilTags and read x, y, and z data from them as shown in the Figures in the Sensing and Control Section above. From this, with itself as the origin the camera was able to read the AprilTag and evaluate its distance and angle from the ankle. The Simulink system was then able to take these values and feed them into its loop module as positioning and movement data for the Arm Control and Movement Control.

We were able to validate the functionality of our actuators (2 servo motors, 1 stepper motor, and 2 DC motors) through independently writing a python script and running it to our chassis. The servo motors were able to move the arm to the specific angles designated by the system, while the linear actuator was able to extend fully.

The results of our Simulink simulation is encompassed by these graphs:



In these graphs, you can see the simulated signals that would be sent to each actuator based on the signals received from the sensors.
In all of our tests, the robot performed as expected with respect to manual and model control. However, not having full system integration with the Pi camera means that we can't validate Chris's expected behavior as an ankle grabbing robot.

Failure analysis and limitations:

Ultimately, our robot not fully working as intended came down to technical difficulties. Our Simulink model ran perfectly in simulation and was completely capable of being implemented in hardware. Additionally, our chassis and arm were completely assembled, mobile, meshed well with all our sensors and actuators, and was able to bear the weight of all parts and movement dynamics after a few modifications. However, we ran into an issue with compilation to the Raspberry Pi, covering:

- Raspberry Pi Camera integration not compiling properly with the perception module
- The open-source Peter Corke Simulink toolbox not being completely compatible with the Raspberry Pi
- Ambiguous networking errors

Our main troubleshooting technique came down to isolating blocks that we knew worked from previous versions and isolated tests of new features. When addressing errors with our blocks, we troubleshooted by probing Simulink blocks outputs and making appropriate changes. By having display outputs on our Simulink model, we can assess unintended movements behaviors without having to test on the actual model. With more time, we would love to continue to integrate more sensors and redesign parts of our Simulink model to continue to debug and integrate our Pi Camera module.

Appendix (Kinematics)

Appendix A

Christopher.m:

```
clear all;
clc;
q = [0 0 0];
d2 = 187.908;
L(1) = Link('d', 67.281, 'a', 0, 'alpha', pi/2,  'qlim', [-pi pi]);
L(2) = Link('prismatic', 'offset', 375.781, 'a', 0, 'alpha', -pi/2, ...
          'theta', pi/2, 'qlim', [0 d2]);
L(3) = Link('d', 0, 'a', 78.2975, 'alpha', -pi/2, 'qlim', [-pi 0]);
robot = SerialLink(L, 'name', 'Christopher');
robot.display();
robot.teach();
```

Appendix B

[utilizing the definitions above from Appendix A]

Kinematics_prop.m:

```
% Forward kinematics:
function T = linkTransform(link, q_value)
  alpha = link.alpha;
  a     = link.a;
  if link.isprismatic
      % prismatic: q_value = d; theta is fixed
      theta = link.theta;
      d = link.offset + q_value;
  else
      % revolute joint: q_value = theta; d is fixed
      theta = q_value;
      d = link.d;
  end
  % DH Transformation matrix
  T = [cos(theta) -sin(theta)*cos(alpha)  sin(theta)*sin(alpha) a*cos(theta);
       sin(theta)  cos(theta)*cos(alpha) -cos(theta)*sin(alpha) a*sin(theta);
       0           sin(alpha)             cos(alpha)            d;
       0           0                      0                     1];
end
A1 = linkTransform(L(1), q(1));
A2 = linkTransform(L(2), q(2));
A3 = linkTransform(L(3), q(3));
T_computed = A1*A2*A3;
fprintf("Our computed transformation matrix for the forward kinematics of our
robot is: \n");
disp(T_computed);
% checking against fkine of the Robotics Toolbox
T_toolbox = robot.fkine(q);
fprintf("Checking against fkine of the Robotics Toolbox:");
T_toolbox.display();
```

```
% Display difference
fprintf("\nDifference between computed and toolbox: \n");
disp(T_computed - T_toolbox.T);   % need the .T to convert to a matrix
```

Appendix C

Workspace_config.m:

```
% Workspace Figure:
------------------------------------------------
N = 5000;  % increase for denser cloud
n = robot.n;  % should be 4
% Pull qlim directly from your link definitions
qlim = zeros(n, 2);
for i = 1:n
   qlim(i, :) = L(i).qlim;
end
% Link 1 has qlim = [0,0] so it's fixed — no variation
Q = zeros(N, n);
for i = 1:n
   range = qlim(i,2) - qlim(i,1);
   if range == 0
       Q(:,i) = qlim(i,1);  % fixed joint, always the same value
   else
       Q(:,i) = qlim(i,1) + range .* rand(N, 1);
   end
end
% Compute end-effector positions via fkine
P = zeros(N, 3);
for i = 1:N
   T = robot.fkine(Q(i,:));
   P(i,:) = T.t';
end
% Plot the workspace
figure;
scatter3(P(:,1), P(:,2), P(:,3), 5, P(:,3), 'filled');  % color
by Z height
colorbar;
xlabel('X (m)'); ylabel('Y (m)'); zlabel('Z (m)');
title('Reachable Workspace of Christopher');
grid on; axis equal;

% Displaying in different configurations
ws = [-700 700 -700 700 -100 400];
```

```
robot1 = SerialLink(L, 'name', 'Config1');
robot2 = SerialLink(L, 'name', 'Config2');
robot3 = SerialLink(L, 'name', 'Config3');
figure(1); robot1.plot([0, 0, 0],           'workspace', ws);
figure(2); robot2.plot([pi/2, 100, -pi/4],  'workspace', ws);
figure(3); robot3.plot([-pi/4, 50, -pi/3],  'workspace', ws);
```

Appendix (Inverse Kinematics and Jacobian)

Appendix A

Inverse_and_Jacobian.m:

```
clear all;
clc;
d2 = 187.908;
L(1) = Link('d', 67.281, 'a', 0, 'alpha', pi/2,  'qlim', [-pi pi]);
L(2) = Link('prismatic', 'offset', 375.781, 'a', 0, 'alpha', -pi/2, ...
          'theta', pi/2, 'qlim', [0 d2]);
L(3) = Link('d', 0, 'a', 78.2975, 'alpha', -pi/2, 'qlim', [-pi 0]);
robot = SerialLink(L, 'name', 'Christopher');
robot.display();
% Generate reachable target poses from known joint configs via FK
q_v1 = [deg2rad(30),  100, deg2rad(-60)];
q_v2 = [deg2rad(-45), 150, deg2rad(-90)];
T_v1 = robot.fkine(q_v1)
T_v2 = robot.fkine(q_v2)
targets = {
   T_v1, q_v1, 'VALID 1';
   T_v2, q_v2, 'VALID 2';
   transl(1000, 1000, 1000), [], 'INVALID';
};
q0 = zeros(1, robot.n);
for i = 1:3
   T_des  = targets{i,1};
   q_true = targets{i,2};
   label  = targets{i,3};
   fprintf('\n=======================================\n');
   fprintf('%s\n', label);
   fprintf('=======================================\n');
   q_sol = robot.ikcon(T_des, q0);
   if isempty(q_sol) || size(q_sol,2) ~= robot.n
       fprintf('Result: FAILED — target outside reachable workspace\n');
       continue;
   end
   T_sol   = robot.fkine(q_sol);
   p_des   = transl(T_des);
   p_sol   = transl(T_sol);
   pos_err = norm(p_des - p_sol);
   if ~isempty(q_true)
       fprintf('q_true:  q1=%.2f deg | q2=%.4f mm | q3=%.2f deg\n', ...
           rad2deg(q_true(1)), q_true(2), rad2deg(q_true(3)));
```

```
    end
    fprintf('q_sol:   q1=%.2f deg | q2=%.4f mm | q3=%.2f deg\n', ...
        rad2deg(q_sol(1)), q_sol(2), rad2deg(q_sol(3)));
    fprintf('Position error: %.4e mm\n', pos_err);
    if pos_err < 1
        fprintf('Result: SUCCESS\n');
    else
        fprintf('Result: FAILED — target outside reachable workspace\n');
    end
end
robot.teach()
J = robot.jacob0(q_v1);
% Differential motion via Jacobian
T = robot.fkine(q_v1);
p = transl(T);
% Desired small end-effector motion
dx = [2.0;   % dx (mm)
      0.0;   % dy
      1.0;   % dz
      0.0;   % dRx
      0.0;   % dRy
      0.0];  % dRz
% Damped least-squares pseudoinverse
lambda = 1e-3;
J_pinv = J' * inv(J*J' + lambda^2 * eye(6));
% Compute joint update
dq = J_pinv * dx;
% Apply small step
alpha = 1.0;
q_new = q_v1 + alpha * dq';
% Check new pose
T_new = robot.fkine(q_new);
p_new = transl(T_new);
disp('Original position:'); disp(p');
disp('New position:'); disp(p_new');
disp('Position change:'); disp((p_new - p)');
% Visualization
figure;
robot.plot(q_v1);
hold on;
robot.plot(q_new);
```

Appendix B

combined_waypoint.m:

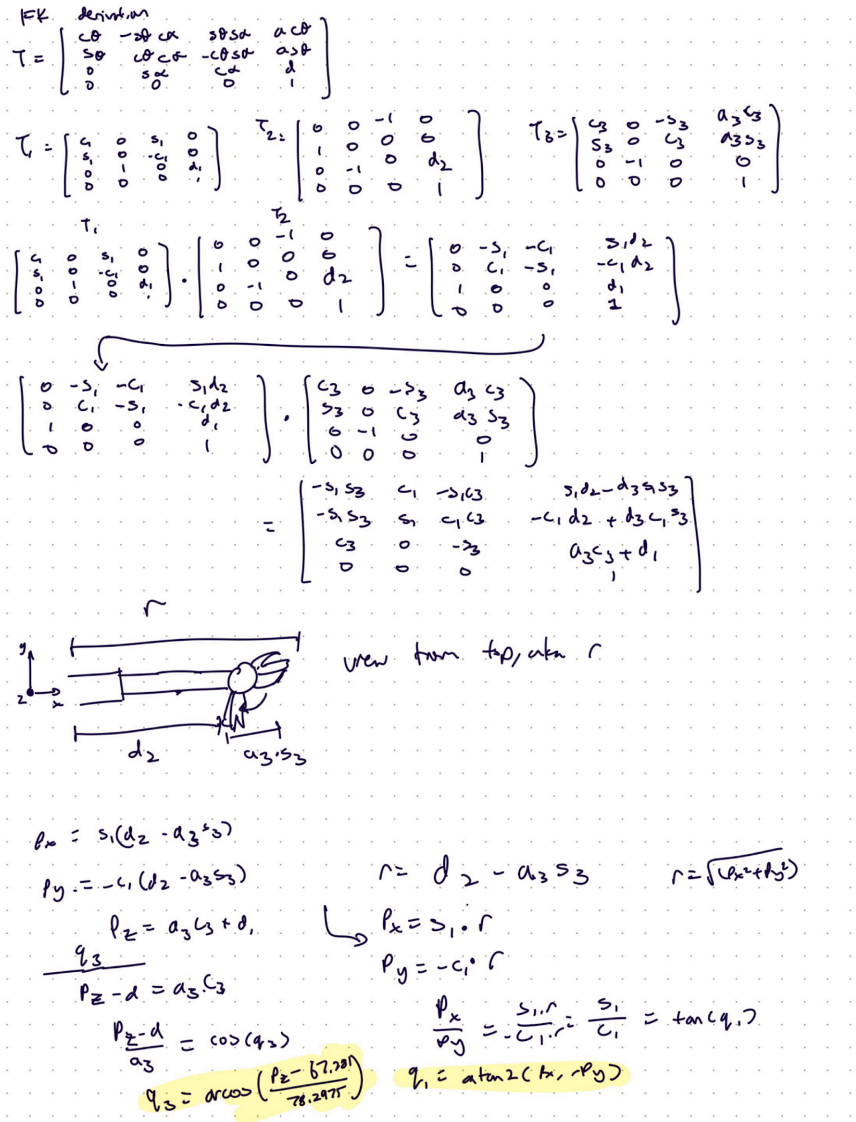
```
clear; clc; close all;
%% Christopher Robot Definition
q0 = [0, 0, 0];
```

```
L(1) = Link('d', 0, 'a', 0, 'alpha', pi/2, 'qlim', [0 pi]);
L(2) = Link('prismatic', 'offset', 2, 'a', 0, 'alpha', -pi/2, 'theta', pi/2,
'qlim', [0 2]);
L(3) = Link('d', 0, 'a', 0.5, 'alpha', -pi/2, 'qlim', [-pi 0]);
robot = SerialLink(L, 'name', 'RPR Robot');
t_half = 0:0.05:1;    %to split the two paths of the joint space trajectories
T1 = transl(0.5, 0.3, 0.25);
T2 = transl(0.5, -0.3, 0.25);
%% Inverse Kinematics
ikopt = {'mask', [1 1 1 0 0 0]};
q1 = robot.ikine(T1, q0, ikopt{:});
q2 = robot.ikine(T2, q1, ikopt{:});  % use q1 as initial guess for q2
%% Joint Space 1 (T1 -> T2)
q_joint1 = jtraj(q1, q2, t_half);
T_joint1 = robot.fkine(q_joint1);
if isa(T_joint1, 'SE3')
   P_joint1 = T_joint1.transl;
else
   P_joint1 = squeeze(T_joint1(1:3,4,:))';
end
%% Joint Space 2 - manual arc OVER the blue line in Z --> to specifically not
follow the path of the red arch
n = length(t_half);
theta = linspace(0, pi, n);
P_joint2 = zeros(n, 3);                          % preallocate nx3 matrix
P_joint2(:,1) = 0.5;                             % X stays fixed
P_joint2(:,2) = 0.3 - 0.6 * (theta/pi)';         % Y sweeps from 0.3 to -0.3
P_joint2(:,3) = 0.25 + 0.3 * sin(theta)';        % Z arches up and back down
%% Cartesian Space (T1 -> T2)
Tc = ctraj(T1, T2, length(t_half));
if isa(Tc, 'SE3')
   P_cart = Tc.transl;
else
   P_cart = squeeze(Tc(1:3,4,:))';
end
%% Plotting
plot3(P_joint1(:,1), P_joint1(:,2), P_joint1(:,3), 'r--', 'LineWidth', 3); hold
on;
plot3(P_joint2(:,1), P_joint2(:,2), P_joint2(:,3), 'g--', 'LineWidth', 3);
```

```
plot3(P_cart(:,1),   P_cart(:,2),   P_cart(:,3),   'b-',  'LineWidth', 3);
waypoints = [0.5 0.3 0.25; 0.5 -0.3 0.25];
scatter3(waypoints(:,1), waypoints(:,2), waypoints(:,3), 100, 'w', 'filled');
grid on;
legend('Complete Joint Space (T1→T2)', 'Cartesian (T1→T2)', 'Waypoints');
title('Complete Joint Space - Christopher Robot');
xlabel('X'); ylabel('Y'); zlabel('Z');
```

Appendix C:

Forward Kinematics to derive equations, inverse kinematics to derive analytical solutions.



Geometric Jacobian derivation: