Catcher Arm

Manuel Gonzalez Parra, Justin Lee, Trevor Oshiro, and Eli Whitaker*

***Abstract*—This paper presents the design, implementation, and experimental evaluation of Catcher Arm, a planar 2-R manipulator that autonomously intercepts airborne ping-pong balls within a 0.30 m x 0.30 m workspace. Two Dynamixel M28-AR actuators and lightweight 3-D-printed links form a 0.6 m-long human-scale arm whose hoop-net end-effector scoops the ball. A stereo global-shutter camera mounted on a rigid 8030 frame streams >100 Hz images to ROS 2 nodes. Computer-vision algorithms threshold, morphologically filter, and contour the ball; Green-theorem moments yield centroids, while stereo disparity supplies depth. Infinite impulse response filters smooth the resulting 3-D trajectory and predict the landing point. Four progressively sophisticated controllers were investigated: (i) independent joint PID with anti-windup, (ii) joint space inverse dynamics, (iii) operational-space inverse dynamics, and (iv) a robust sliding-mode variant further mitigated model error. System parameters were tuned through empirical step-response trials, targeting a 0.5 s settling time demanded by projectile kinematics. Experiments demonstrate reliable catches over a spectrum of ball velocities and initial arm poses, confirming that centralized dynamic compensation significantly outperforms decentralized position control.**

***Index Terms*—Robot Arm, Projectile, Interception, 2-DOF, Computer Vision**

I. Overview

The robotic system chosen for this project was a custom 2R manipulator that operates in the horizontal plane with the goal of catching a ping-pong ball in the air. The motivation behind this project came from the initial interest in building a juggling robot, a goal that would be far too ambitious for the duration of this course. Thus, the subtask of focusing just on the catching aspect seemed like a natural first step. Catching a ball on its own is a challenging task on its own due to the relatively high speeds and large workspace involved. This robot aims to push the limits of the Dynamixel kits that were given to us to inspire future teams to build more dynamically ambitious projects while teaching us about the advantages and disadvantages of centralized control compared to the decentralized control with which industry is more familiar.

II. Related Work

Multiple papers have been written detailing similar efforts to create robotic arms capable of catching projectiles. Several of these documents were reviewed to provide insight and foundational background information.

A. Catch It! Learning to Catch in Flight with Mobile Dexterous Hands

This paper [1] describes an effort to create a ball catching system consisting of a 6-DOF arm and 12-DOF hand mounted to a mobile base. It places significant emphasis on an architecture that splits the process into two subtasks, as well as reinforcement learning based training to create and tune its control policy. It also details various challenges faced such as difficulties with camera perspective transformations, excessive degrees of freedom, and optimization, as well as the approaches that were used to address these issues. While many of the complexities detailed in this paper such as machine learning and sim2real transfer are outside the scope of our project, many of the involved approaches can be abstracted and applied within our context. Coordination between both the mobile cart and the arm's control scheme are analogous to the centralized control we set as a target for our system, and their use of a two-subtask division highlights the effectiveness of an architecture that reacts very quickly to move into a rough position before dialing in its precision from there. Similarly, their training reward and penalty parameters mirror goals for our system such as both the value of positional accuracy as well as the undesirability of excessive control efforts that may be beyond our actuators viability.

B. Unified Control Framework for Real-Time Interception and Obstacle Avoidance of Fast-Moving Objects with Diffusion Variational Autoencoder

Apan Dastider et al. [2] propose a unified control framework that lets a 7-DoF arm intercept high-speed objects while dodging obstacles in real time. Their key idea is to compress the robot-obstacle state (18 variables) into a smooth two-dimensional manifold using a diffusion variational auto-encoder (D-VAE). An offline, densely connected graph built in this latent space enables rapid shortest-path searches, producing collision-free joint targets. Online, an extended Kalman filter tracks the object's 3D trajectory and triggers graph re-planning whenever the target or obstacles move. Decoding the latent path back into joint commands yields smooth interception motions that succeed in 94% of simulated and physical trials, with planning times under six seconds - far faster than RRT, MPNet or other baselines. Although this paper describes techniques far beyond those attempted in our computer vision and control systems, the extended Kalman filter (EKF) was a compelling option that was considered for the ball-predictor node that tracks and predicts where the ball would land. Due to time constraints, we instead implemented a simpler infinite impulse response filter. Like the EKF, the infinite impulse response filter uses current and past position information for the prediction of future ball locations, reflecting similar goals of balancing accuracy and responsiveness under noisy measurement conditions.

*All authors are with the Henry Samueli School of Engineering, University of California at Los Angeles

C. Kinematically Optimal Catching a Flying Ball with a Hand-Arm System

This source [3] exhibits their contributions to developing a ball catching robot system that utilizes a 7-DOF arm and 12-DOF four-fingered gripper. Compared with previous systems, this arm utilizes the dexterous gripper system to implement a robust mechatronic system for catching at the end effector. In the control side, the research with this arm involves a non-linear optimization of the catch point selection, arm configuration selection, and path generation in one process. During the operation, the robot takes in a given initial configuration/robot state, and attempts to intercept the ball trajectory given predicted with the vision system and path selection from methods like probabilistic roadmaps and rapidly exploring trees. The observed limitations within the system were mainly derived from joint speed limits (when choosing appropriately limited accelerations) and the overall vision system, as computationally efficient computational methods were implemented with parallel computation on a cluster of 32 CPU cores. Their resulting catch rate from the system was over 80%. While on a higher level of dexterity, this system from their research gives insight to the general implementation of catching methods for robots. The dynamic motion of the ball during its trajectory would necessitate higher velocities of the joints themselves, indicating the joint speeds to also be a limiting factor for this project. Within the vision system itself, there was a need for re-computing the predicted trajectory of the ball itself that likely caused issues in accuracy of motion for the robot in this article. Within this project, the limitations imposed with the vision system were also explored through the various implementation of filters and trajectory calculation methods.

D. A Dynamical System Approach for Catching Softly a Flying Object: Theory and Experiment

This paper [4] discusses the challenges associated with dynamic motions, particularly object catching. The main issue deals with the precision of the motors and the trajectory information, as any small imprecisions could cause the end effector to close too early or late and not securely catch the projectile by the time it reaches it. Their proposed solution involves what they describe as "soft catching", where the end effector temporarily moves along with the projectile before any gripping or closing motions begin. The idea is that the ball will have more time in the gripping range of the end effector throughout its motion giving it more time for a successful grab. The paper focuses on three criteria for the motor: it must be at the right time, place, and velocity to catch the object softly. Their method is based on a second-order Linear Parameter Varying dynamical system, and they discuss their use of Gaussian Mixture Regression to estimate its parameters. Although this paper discusses the catching of a projectile in 3D space, it offers insight into the methods and criteria needed for our planar manipulator. Since our manipulator relies on only matching the planar position of the projectile, the diameter of the hoop and speed of the ball necessitates a more precise and responsive approach to be taken to achieve successful catches. The paper emphasizes velocity alignment and an adaptive intercept point estimation, and reports a 71.6% success rate using soft catching compared to a 1.6% success with hard catching. Understanding and applying the principles discussed in this paper could help us improve the reliability and responsiveness of our implemented controller.

III. Objectives of Controller Design

As shown in Fig. 1, the ball's position is tracked with a high-speed, stereo, global shutter camera mounted to an 8020 rail structure that rises 3 feet above the arm's workspace. After performing simulations with the rated capabilities of the actuators and some preliminary testing, we came to the conclusion that limiting our workspace to about 1 ft by 1 ft would give us the best chance at catching a ping pong ball with enough time buffer for computational latency. This project was designed with centralized control in mind due to its high speeds and moments of inertia that would amplify nonlinear effects. The controller would need to be both robust and simple enough to handle noise in the positional measurements of the ball by the vision system and to be easily computed in real-time.

IV. Methods

A. Mechanical Design

Our robotic system, shown in Fig. 1, is composed of the robotic arm and camera structure both mounted to a steel panel for consistency between the camera coordinate system and the arm coordinate system. The arm consists of the two Dynamixel M28-AR actuators and four 3D-printed parts: the base that rigidly joins the first actuator to the steel panel, the two upper-arm link pieces that join the first actuator output flange to the second actuator's housing, and the forearm link that also has a hoop with a net as its end effector. The camera structure is composed of two 8020 rails (and brackets), the 3D-printed camera mount, and the camera module. In order to cover an area of 1ft by 1ft, the arm links were sized to 1 ft each, resulting in an arm that is similar to that of a human in length. Figure 19 in the appendix shows the mechanical assembly of the system made within Onshape, and figure 1 shows the physical setup of the hardware for the system.

B. System Considerations

A robust operational space inverse dynamics control was envisioned to be the best suited for this application since we were considering taking advantage of the ball's constant horizontal velocity to predict where it would be headed in order to better account for latency. The justification for this controller architecture was that it would allow us to more easily follow the linear trajectory of the ball instead of creating trajectories that would ultimately reach the same destination but wouldn't match the path the ball was taking. This relies on several assumptions such as minimal air resistance that would slow down the ball or change its direction mid-flight, accurate measurements of the ball in space and time such that

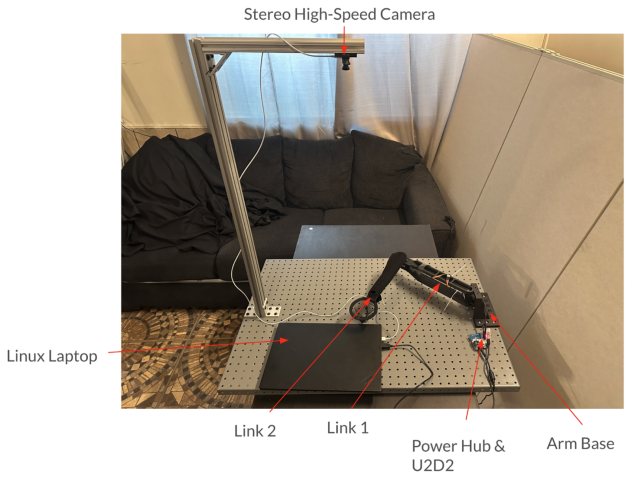


Fig. 1: Experimental Setup

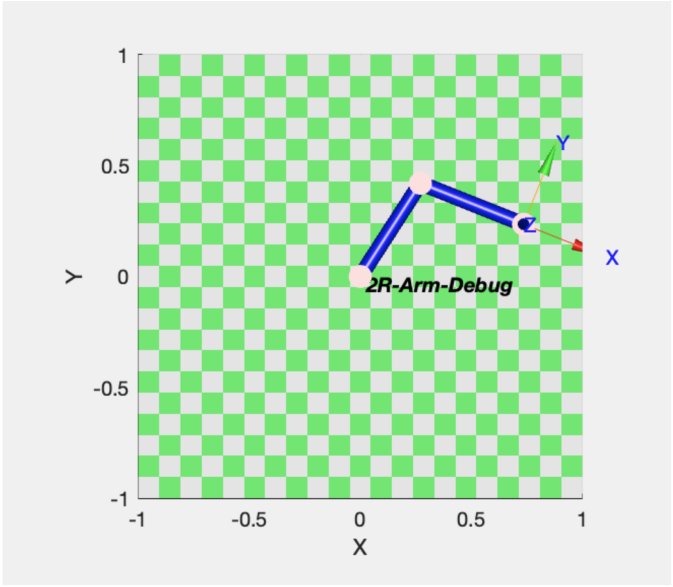


Fig. 2: Visualization of Arm in Matlab

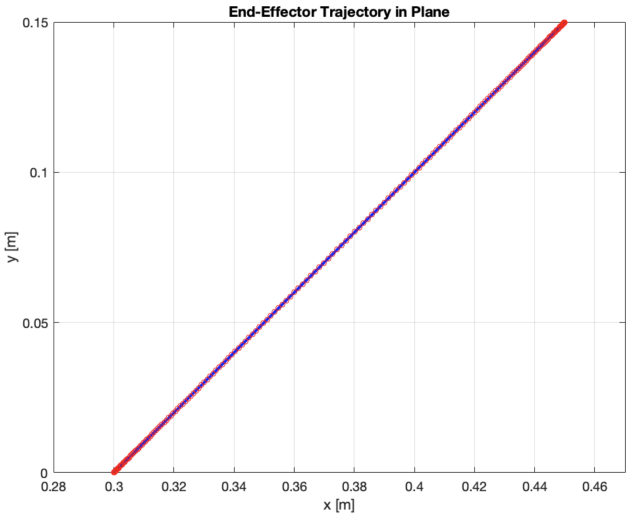


Fig. 3: End Effector Trajectory in Matlab

the generated trajectories follow the path of the bal, and a sufficiently accurate model that could cancel out the nonlinear terms of the manipulator equation.

C. Design Requirements

In order to reach the falling ball in time before falling below the end effector we estimated that the arm would need to have a settling time of approximately 0.5 s. Initial simulations in matlab, arm shown in Fig. 2, using a conservative physical model of the arm and the actuator torques and speeds allowed us to be confident that the arm was physically capable of reaching opposite corners of the workspace quickly, as shown in Fig. 3 without requiring any more torque or speed than the actuators could provide. Plots that show the joint kinematics and torques are shown in Figs. 4 and 5. Ultimately, the success of the arm is determined by how well it is able to catch a ping pong ball in various scenarios such as changing the starting position of the arm, the location of the ball bounce, the velocity of the ball, etc.

D. Software Design

This robotic system uses ROS 2 for the communication between the camera system and the arm. There are 4 main nodes that allow the arm to perform: the gscam_node that bridges between GStreamer and ROS 2, the ball_predictor_node which performs the computer vision and determines where the ball will land based on its measured kinematics, the pixel_to_xy_bridge that converts pixels to coordinates using real units that can be mapped onto the taskspace, and finally the controller that performs the inverse kinematics and outputs either a simple position command to the actuators or a PWM signal based on the specific type of controller designed. The vision system is only used to track the ball and predict its landing position while the onboard encoders are used to track the state of the arm. A layout and mapping of these nodes is shown in Fig. 6, and shows how the eosystem is constructed.

E. Computer Vision

Erosion and Dilation. To detect the ball, we used the `OpenCV` library [8]. A threshold of 225 out of 255 was used to let the functions know what pixels to consider "black" and what pixels to consider "white":

$$m(x,y) = \begin{cases} 255, & g(x,y) \geq T, \\ 0, & g(x,y) < T. \end{cases} \quad (1)$$

Erosion and dilation are functions that remove noise such as small white speckles and fill in any voids or jagged edges. This is done by looking at each pixel's eight neighbours and eroding it if atleast one of its neighbors is below the threshold

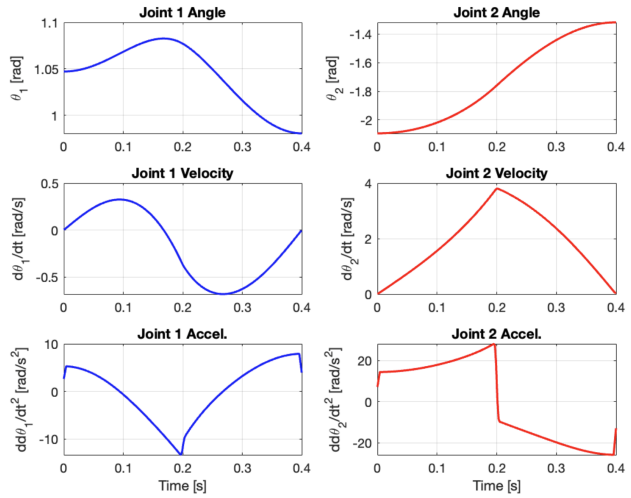


Fig. 4: Joint Kinematics in Matlab Simulation

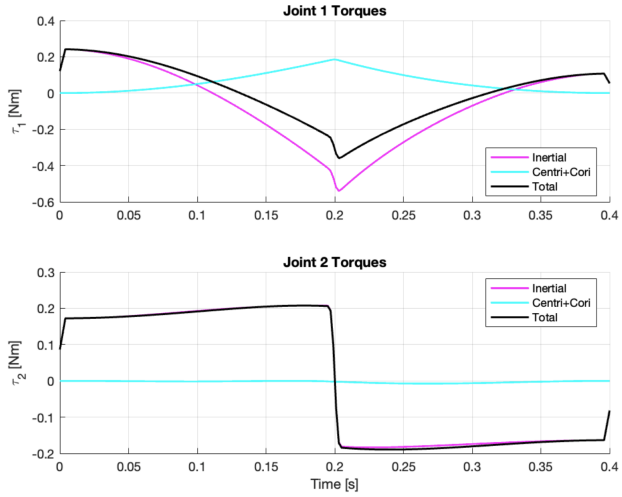


Fig. 5: Joint Torques in Matlab Simulation

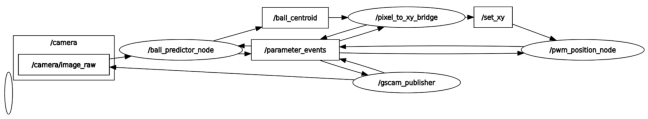


Fig. 6: Nodes and Topics used in ROS 2

and dilating it if at least one of its neighbors is above the threshold, as expressed by

$$e(i,j) = \min_{(u,v)\in K} m(i+u, j+v) = \bigwedge_{(u,v)\in K} m(i+u, j+v), \quad (2)$$

$$o(i,j) = \max_{(u,v)\in K} e(i-u, j-v) = \bigvee_{(u,v)\in K} e(i-u, j-v). \quad (3)$$

Composed together, these form the opening operation

$$m_{\text{open}}(i,j) = (m \ominus K) \oplus K, \quad (4)$$

where $\ominus$ and $\oplus$ denote the morphological erosion and dilation, respectively.

Contour extraction uses the Suzuki–Abe Border–Following Algorithm [9] to find the edges of the white regions in the image, where the largest white region should correspond to the ball. The general outline of how the algorithm works is to

1) Scan the image row by row and add white pixels that are not already part of a contour to a new contour.
2) Initialise at starting pixel p_0 with previous search direction $d_0 = 7$, which corresponds to coming from the left.
3) At step k, you are at pixel p_k with previous search direction d_k. The eight surrounding neighbours are examined in the order $\big[(d_k+6) \bmod 8, (d_k+7) \bmod 8, (d_k+1) \bmod 8, \ldots, (d_k+5) \bmod 8\big]$, where each direction d is a natural number from 0 to 7 corresponding to an offset of a pixel. The first neighbour that is white becomes p_{k+1}, and is appended to the contour.
4) Stop once you return to p_0, which guarantees one full loop.

Moment computation via Green's theorem. Green's theorem can be used to compute information about the interior of a region just by using the outer boundary:

$$\oint_C x\,dy - y\,dx = 2 \iint_{\text{inside}(C)} dA. \quad (5)$$

Specifically, it can be used to find the zeroth and first moments of the blob whose quotient is the centroid of the ball. In continuous form

$$m_{p,q} = \iint_{(x,y)\in\text{inside(contour)}} x^p y^q\, dA. \quad (6)$$

Applying Green's theorem [10] in discrete form reduces the zeroth moment to

$$m_{0,0} = A = \frac{1}{2} \sum_{i=0}^{N-1} \Delta_i, \quad (7)$$

while the first moments become

$$m_{1,0} = \frac{1}{6} \sum_{i=0}^{N-1} (x_i + x_{i+1})\,\Delta_i, \qquad m_{0,1} = \frac{1}{6} \sum_{i=0}^{N-1} (y_i + y_{i+1})\,\Delta_i. \quad (8)$$

The centroid of the blob is therefore

$$c_x = \frac{m_{1,0}}{m_{0,0}}, \qquad c_y = \frac{m_{0,1}}{m_{0,0}}. \quad (9)$$

Stereo depth estimation. Using the distance between the two camera lenses (the baseline B) and the offset of the centroids in the two images (the disparity d), depth is obtained via similar-triangle geometry [11]:

$$Z = \frac{Bf}{d}, \quad (10)$$

where f is the focal length in pixels.

Once Z is known, the 3-D position of the ball can be determined and thus its projectile kinematics. Although this method was far faster than tracking the ball throughout its

trajectory, it was also significantly noisier, leading to instability when differentiating to obtain velocity. Holding the ball still below the camera produced readings that could vary by as much as a full centimetre depending on depth, corresponding to metre-per-second velocity errors at 100 Hz.

Noise suppression with an IIR filter. An economical solution was to implement an infinite-impulse-response (IIR) low-pass filter that smoothed out the noise:

$$y_n = \alpha y_{n-1} + (1-\alpha)x_n, \qquad \alpha \in [0,1], \tag{11}$$

where y is the filtered value of the raw measurement x. These filters were applied to the 3-D position and velocity estimates of the ball, as well as to the predicted landing site. Because the IIR filter incorporates the entire history of the trajectory, a rough estimate of the landing location can be produced immediately and refined as additional frames arrive.

F. Dynamixel Position Control

The first control scheme that was attempted was to use the actuators' onboard PID controllers in position control mode so that the only computation required would be of the inverse kinematics that would take in cartesian positions in the workspace and return joint positions for the actuators to move to. The classic 2-Link arm inverse kinematic equations were successfully utilized and are shown below, where r refers to the distance between the base origin of the arm and the command (x,y) position.

$$c_2 = \frac{r^2 - L_1^2 - L_2^2}{2L_1L_2}, \tag{12}$$

$$s_2 = \sqrt{1-c_2^2}, \tag{13}$$

$$\theta_2 = \text{atan2}(s_2, c_2), \tag{14}$$

$$\theta_1 = \text{atan2}(y,x) \; - \; \text{atan2}\big(L_2 s_2,\, L_1 + L_2 c_2\big). \tag{15}$$

Because of this, the ROS node was called the `inverse_kinematics_node` as a proof of concept that would show that the arm's inverse kinematics was working correctly and to get a sense for the physical capabilities of the arm before tuning the actuators. Below is the implementation of the inverse kinematics in our C++ code.

G. Independent Joint Control

Independent Joint Control

Once the arm and camera were well calibrated such that pixels matched with coordinates for the arm inverse kinematics through the onboard position control of the actuators, we decided to tune the PID gains by implementing our own controller in a ROS 2 node and adding an anti-windup clamp. The control law is the familiar equation shown below:

$$u \; = \; K_p e \; + \; K_i \int e\,dt \; + \; K_d \dot{e}, \tag{16}$$

which attempts to minimize the measured error e by applying three actions: the proportional action, the integral action, and the derivative action. The proportional action is, as the name implies, proportional to the error, so as the joint output diverges more from the commanded joint position the proportional controller will increase to counteract that. The integral action is used to reduce steady-state errors by allowing these small errors to accumulate and eventually produce enough action to correct the system. The derivative action acts as a damper, slowing down the rate of change of the error so that the output, for example, doesn't overshoot.

H. Inverse Dynamics Control

Inverse Dynamics Control

Once we could reliably control the arm with our own PWM signal via the PID controller, we moved on to a simple implementation of centralized control: inverse dynamics control as shown in Fig. 7. This required constructing a model of our physical system that included the masses and moments of inertia of the manipulator. The following control law is used to linearize and decouple the dynamic relationship [5]:

$$u \; = \; By \; + \; n, \tag{17}$$

where

$$n = C\dot{q} \; + \; F\dot{q} \; + \; g, \tag{18}$$

$$y = -K_p \tilde{q} \; - \; K_d \dot{\tilde{q}} \; + \; r. \tag{19}$$

Here, q is the vector of joint positions, $\tilde{q}$ is the error with respect to the desired trajectory, and r is an appropriate reference term to generate the desired dynamics in joint space. By applying the manipulator model to cancel its nonlinear effects on the right-hand side, the remaining system on the left-hand side is linear and can be stabilized by proportional and derivative actions. The resulting error dynamics satisfy the second-order equation:

$$\ddot{\tilde{q}} \; + \; K_d \dot{\tilde{q}} \; + \; K_p \tilde{q} \; = \; 0, \tag{20}$$

which is asymptotically stable for positive-definite matrices K_p and K_d.

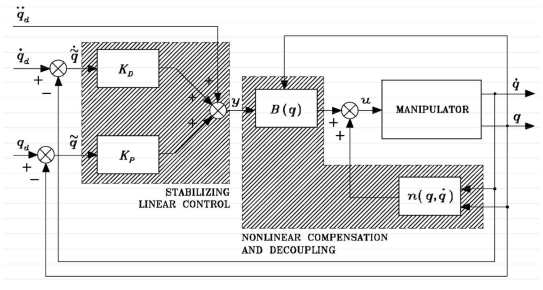


Fig. 7: Block diagram of operational space inverse dynamics control.

I. Operational Space Inverse Dynamics Control

Operational Space Inverse Dynamics Control

Similar to the previous controller, operational space inverse dynamics (OSIDC) control takes advantage of the manipulator dynamic model to compensate for the nonlinear and coupling effects between the joints; however, it does so in the operational (task) space x instead of joint space q. The overall control law has the same structure [5]:

$$u = By + n, \tag{21}$$

but the command y now accounts for the mapping to operational space. Specifically:

$$y = J_A^{-1}\left(-K_p\tilde{x} - K_d\dot{\tilde{x}} + \ddot{x}_d - \dot{J}_A\dot{q}\right), \tag{22}$$

where:

- x is the end-effector/task-space position vector,
- $\tilde{x} = x - x_d$ is the task-space error with respect to the desired trajectory x_d,
- $\dot{\tilde{x}} = \dot{x} - \dot{x}_d$ and $\ddot{x}_d$ is the desired task-space acceleration,
- $J_A(q)$ is the analytic Jacobian mapping joint velocities $\dot{q}$ to $\dot{x}$,
- $\dot{J}_A(q,\dot{q})$ captures the time derivative of the Jacobian,
- K_p and K_d are positive-definite gain matrices in task space,
- $n = C(q,\dot{q})\dot{q} + F(q,\dot{q})\dot{q} + g(q)$ represents the nonlinear terms (Coriolis/centrifugal, friction, gravity, etc.),
- $B(q)$ is the inertia-related mapping in joint space used to cancel nonlinearities when premultiplying y.

By applying the manipulator model to cancel nonlinear effects, the closed-loop error dynamics in operational space satisfy:

$$\ddot{\tilde{x}} + K_d\dot{\tilde{x}} + K_p\tilde{x} = 0, \tag{23}$$

which is asymptotically stable for positive-definite K_p and K_d.

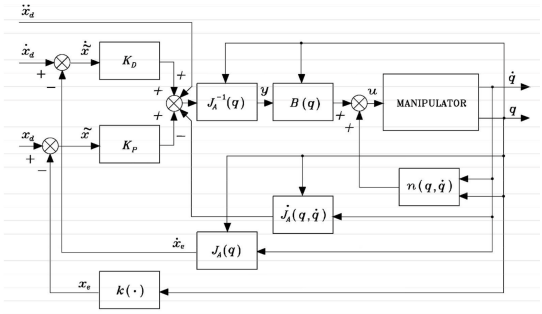


Fig. 8: Block diagram of operational space inverse dynamics control.

One of the main challenges when using operational space inverse dynamics control is tuning the gains in the presence of singularities. Small changes in the desired task-space direction near a singularity can cause large joint velocity commands. The analytic Jacobian J_A and its inverse must therefore be handled with care as they are fundamental to operational space control, as seen in figure 8 (e.g., via damped pseudoinverse or avoidance strategies) to ensure stable and safe operation.

J. Joint Space Robust Control

Robust Control (Sliding Mode)

Unlike the previous two centralized controllers, the robust controller does not rely on the assumption that the dynamic model perfectly compensates the nonlinear terms of the system dynamics. Instead, it defines the control vector as [5]:

$$u = \hat{B}(q)y + \hat{n}(q,\dot{q}), \tag{24}$$

where $\hat{B}(q)$ and $\hat{n}(q,\dot{q})$ are the model estimates used for compensation. Based on the estimates of the model, we define the model errors:

$$\tilde{B} = \hat{B} - B, \tag{25}$$

$$\tilde{n} = \hat{n} - n. \tag{26}$$

The advantage of robust control is that the model can be imperfect, yet disturbances and model mismatches can be suppressed more effectively than with the previous two controllers. The secondary input vector y is given by:

$$y = \ddot{q}_d + K_D\dot{\tilde{q}} + K_P\tilde{q} + w, \tag{27}$$

where:

- q_d is the desired joint trajectory, with acceleration $\ddot{q}_d$,
- $\tilde{q} = q - q_d$ is the joint-position error,
- $\dot{\tilde{q}} = \dot{q} - \dot{q}_d$ is the joint-velocity error,
- K_P and K_D are positive-definite gain matrices in joint space,
- w is the robustness term (sliding-mode injection) defined below.

This robust control scheme, often called sliding mode control, works by attracting the system state to a subspace (the sliding subspace) on which the indeterminacy of the dynamic model is eliminated. The robustness term w that causes this attraction is defined as:

$$w = \frac{\rho}{\|z\|}z, \tag{28}$$

where $\rho > 0$ is a tuning scalar that determines the strength of attraction, and

$$z = D^T Q\xi \tag{29}$$

is a projection of the state error onto a direction orthogonal to the sliding subspace. Here:

- ξ is the state error of the manipulator, typically combining position and velocity error,
- $D^T Q\xi = 0$ defines the sliding subspace onto which the error is initially attracted.

Once on the sliding subspace, the system behaves more similarly to the previous control systems but without requiring a highly accurate model.

By inspecting the block scheme in Fig. 9, it is clear that the robust controller shares much with the joint-space inverse dynamics controller but has the additional robustness that: first, reorients the state error orthogonal to the sliding subspace via $z = D^T Q\xi$; and second, scales it by $\rho/\|z\|$ before contributing it to the secondary control input y. This ensures that even with

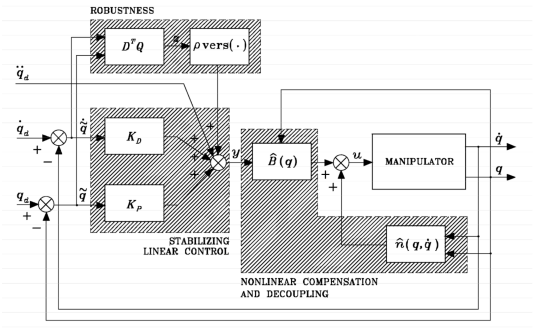


Fig. 9: Block diagram of joint-space robust (sliding mode) control.

model uncertainties, the closed-loop system is driven toward the sliding subspace, where the dynamics become effectively linear and easier to stabilize.

V. Results

To ensure a steady-progression to our ultimate goal, the robotic system's control architecture was iteratively made more complex by passing through simpler implementations, for both the arm control and the vision system. Thus, they both underwent an evolution that allowed for improved performance or at least a deeper understanding of the nuances of controller design.

A. Independent Joint Control

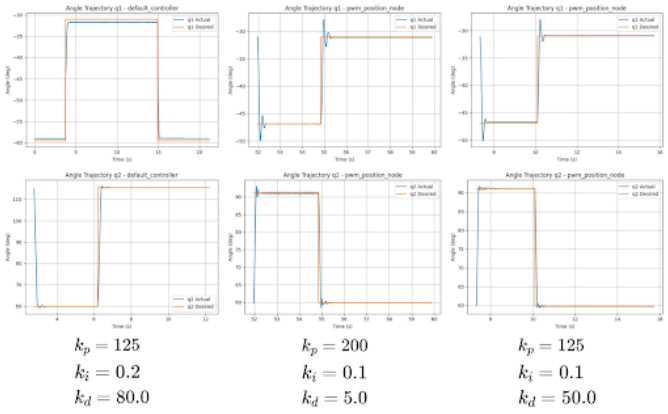


Fig. 10: Step Responses of Decentralized Controller

Within figure 10, the tuned controller can be found on the left most side, and other gain values can be found on the middle and right columns. When starting tuning, higher proportional gain values closer to the middle column were chosen to obtain a fast reaction time for the arm's dynamic movement. However, tuning the derivative and integral gains became difficult and increasingly sensitive for higher gain values. This was likely a result of the speed induced by the PWM signals to the motor saturating as the controller was pushing to the actuator's limits. From there, proportional gain values were decreased while tuning to find the minimal gain value that produced a similar response time to the step inputs.

An intermediate result can be found with the rightmost column in the figure above. The derivative value was then tuned to account for the oscillations seen with the earlier gain values for higher proportional gain. The derivative gain was then tuned more to achieve the final tuned gain values seen with the leftmost column in the figure. Overshoot and oscillations in the response were minimized through the proportional and derivative tuning, and the integral gain was tuned to decrease steady state error. Although a steady state error can be seen within the final tuned plot, this was deemed acceptable for our use case, as the overall position error was within the bounds needed for a successful capture of the ball. Snapshot of the controller in action can be found in figure 11.

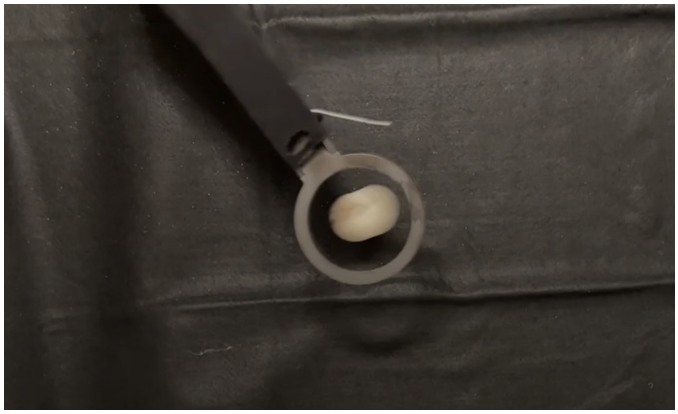


Fig. 11: Image of Decentralized Controller on Hardware

Some additional notes to be made in the process were the fact that the same gain values were used for each actuator. Due to time constraints, this simplification was made for tuning. However, as seen more explicitly with the leftmost and rightmost columns, the resulting response of the actuators had differences in their steady state error and overshoot. In particular, the actuator at the shoulder exhibited more sensitivity to overshoot when subject to higher proportional and lower derivative gains.

B. Joint Space Inverse Dynamics Control

The responses of the joint space inverse dynamics controller can be seen in figure 12. As with the previous controller, the proportional gain value was first tuned to achieve optimal reaction of the system to the ball's dynamic movements. However, tuning this controller took the gain values for the different joints into consideration unlike the decentralized PWM controller. The proportional gain values were first tuned by ramping the gains for both gains up at the same value to identify a gain for one of the joints to achieve adequate response time. The result of this can be seen with the middle column in the figure above, where the response time of the shoulder joint was deemed accurate, and further increase of the elbow joint's proportional gain values was attempted until the rightmost column's gain values were obtained. With these

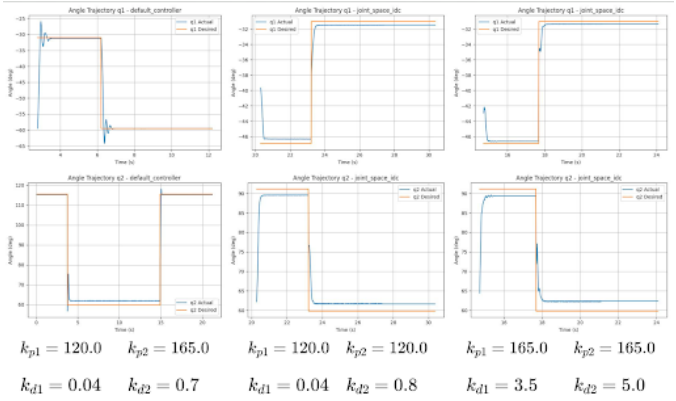


Fig. 12: Step Responses of Joint Space Inverse Dynamics Controller

values of proportional gain, the derivative gain values were then tuned to reach smoother responses of the system.

The tuned gain values for adequate performance can then be seen with the leftmost column in the figure. While the resulting performance for catching was achieved, it can be seen that the oscillation and overshoot behavior of the response increases as compared to some of the result responses seen in the other columns. This was likely due to the process of attempting to identify the proportional and derivative gains for the actuators individually. The inverse dynamics component of the controller accounts for the coupling effects of the dynamic movement of the gains, so the methodology may have not been optimal for this specific controller. Future improvements for tuning this controller would potentially involve a recursive process, where the gains would be further tuned after reaching a benchmark value described with the process previously. This would potentially optimize and reduce the overshoot and oscillation in the response as shown with our final gains for this project. A snapshot of the controller implemented in the hardware system can be seen in 13.

Fig. 13: Image of Joint Space Inverse Dynamics Controller on Hardware

Some additional notes to be made with this project would be the erratic behavior observed when the arm would swing to the side at iterations of testing the controller. This is likely caused by an instance where the computational process of the controller would have an error when the end-effector reaches places that are close to singularities for the system. During testing, this was observed at times where the robot would move towards a coordinate point that was close to full extension of the joints. This was also further backed with the comparison with the decentralized controller, as these cases of critical failure weren't present within the operation of this controller. Another note for the implementation of this system would then also be about the process of obtaining step responses itself.

To achieve the responses presented in the figure above, the robot was commanded to move between two coordinates defined within the task space of the system. All three columns were from cases where the same coordinate in the task space was invoked for the command, which was a simple motion along the x-axis for 10cm. As the centralized controller accommodates for the full dynamics of the system, further testing for different coordinates at potential edge cases of the system would have likely been beneficial for further analysis. Especially for areas closer to singularity, higher responsiveness of the system may not have been achievable with how sensitive the motion of the arm could have potentially been in certain directions.

C. Operational Space Inverse Dynamics Control

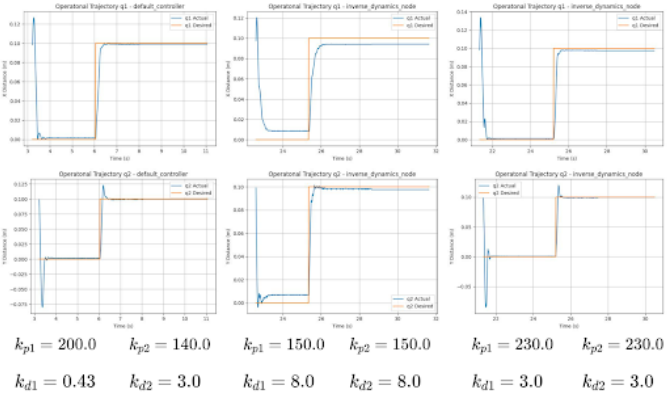


Fig. 14: Step Responses of Operational Space Inverse Dynamics Controller

This controller was similar to the joint space inverse dynamics controller, as the model simplifications and control law operate in a similar process. However, the gains were applied in operational space, as opposed to the task space inverse dynamics control (OSIDC) which applies gains directly to the joints. This process was less intuitive compared to the joint space, as the presence of singularities and overall manipulator geometry made instabilities or sporadic joint torques to become more common. Since the controller was designed in task space, with the inputs and gains both being applied to the cartesian directions, the step responses in figure 14 were also shown in task space both to have a more meaningful display of precision. 10cm goal trajectories were chosen as opposed to joint angle trajectories for OSIDC for this

reason. Responsiveness of the system was prioritized, so the controller's proportional gains were pushed to the maximum value we could use to obtain adequate responses that had minimal overshoot. The derivative gains were then used to obtain smoother responses of the system along coordinate directions.

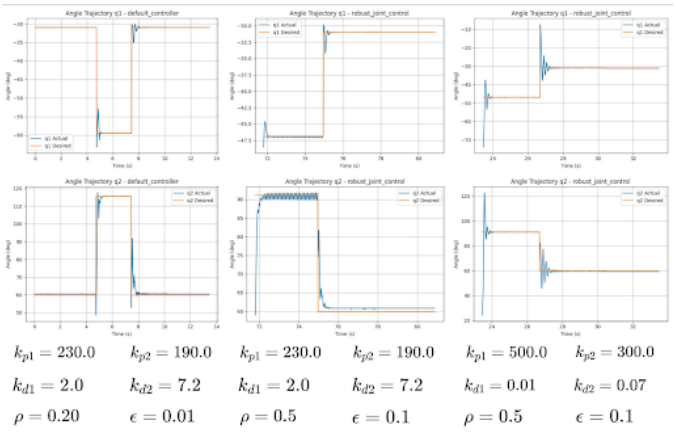
In comparison to the joint space inverse dynamics controller, the tuning of this controller was somewhat more sensitive to changes in gain values. Seen in the step responses for the controllers, the task space inverse dynamics controller exhibited more jittering in its motion relative to the joint space version. Further tuning of the derivative term helped to minimize this behavior, after a general proportional gain range was obtained. The final tuned values shown on the leftmost side of the figure above were obtained after being deemed adequate for the controller's responsiveness to the ball's dynamic movements. An example of the controller in operation on hardware can be seen with figure 15.

Fig. 15: Image of Operational Space Inverse Dynamics Controller on Hardware

During the testing of this controller, the arm seemed to be notably more sensitive to the erratic behavior of swinging to the side during testing. When selecting the step size for testing the arm, this controller was the main factor in determining the value. The system seemed to be more sensitive to larger distance step values input,especially as it meant the end effector would approach a singularity during its motion. The fundamental framework for OSIDC meant that applying gains in task space, particularly when an instantaneous velocity in a specific direction is zero as near a singularity, had to be done conservatively. To effectively characterize the response, the 10cm step size was chosen, as it was observed to be the approximate threshold step value the robot would use to move without any errors. The cause for this error is likely to be the same as for the joint space inverse dynamics controller, as the further distances that created these failures were closer to full extension of the arm joints for singularity conditions.

D. Robust Joint Space Control

Finally, the robust controller plots for step responses can be seen in figure 16. This controller involved tuning of the gains as well as the rho and epsilon parameters as shown below



$k_{p1} = 230.0$	$k_{p2} = 190.0$	$k_{p1} = 230.0$	$k_{p2} = 190.0$	$k_{p1} = 500.0$	$k_{p2} = 300.0$
$k_{d1} = 2.0$	$k_{d2} = 7.2$	$k_{d1} = 2.0$	$k_{d2} = 7.2$	$k_{d1} = 0.01$	$k_{d2} = 0.07$
$\rho = 0.20$	$\epsilon = 0.01$	$\rho = 0.5$	$\epsilon = 0.1$	$\rho = 0.5$	$\epsilon = 0.1$

Fig. 16: Step Responses of Robust Joint Space Controller

the figure. As the impact of the proportional and derivative gains were more intuitive for the system, they were first tuned to achieve adequate responses as shown in the middle and rightmost plots. When tuning the proportional values, the system was found to accept relatively higher proportional gains compared to the inverse dynamics controllers to achieve higher responsiveness in performance, while also successfully converging in its response. Tuning the derivative gains in this controller were also notably more sensitive in managing overshoot and oscillation in the system.

Compared to the inverse dynamics controllers, the robust controller exhibited notably more smooth motion as compared to the other controllers. However, the oscillation motion was more evident as the controller approached and maintained the steady state value from the step, which can be prominently seen in the middle column of the figure above. Further tuning of the robustness parameters shown in the final tuned values to the left then achieved the desired reduction in oscillation for the system. For the epsilon parameter in particular, the increase in value of the parameter showed notably more impact on the resulting oscillation of motion present in the system. The action of the controller can be seen with the image in figure 17.

As this controller was in the joint space, the step response of the system was set to solely movement in the x direction. The robust controller motion also showed less sensitivity to erratic motion of swinging to the side during testing of the arm. However, further testing of different joint steps would help to further establish this observation for the controller. Compared to the inverse dynamics controllers, the frequency of jittering in the motion and maintenance pose was also more noticeable during testing of the controller motion. This had minimal impact on the resulting capture of the ball itself, but the motion itself would potentially make the robot less efficient in its motion.

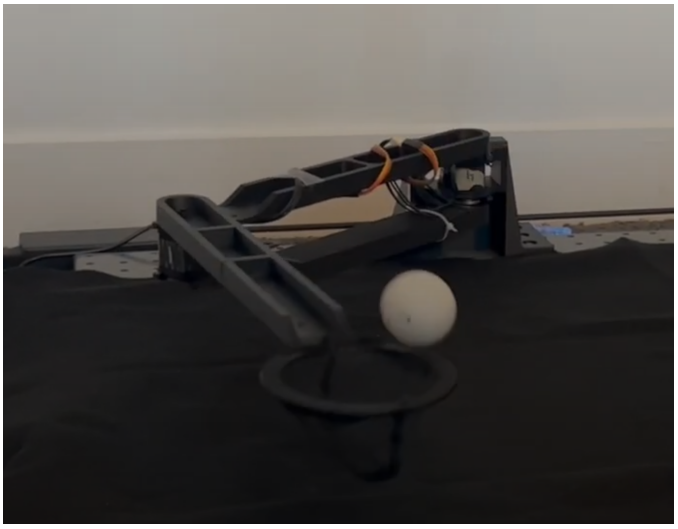


Fig. 17: Image of Robust Joint Space Controller on Hardware

VI. Discussion

The gradual progression in our solution evolving from one controller to the next was spurred by efforts to troubleshoot issues in system performance. While our initial independent joint control was fast and had the capability to catch the ball, the unaccounted-for dynamics of the system could easily saturate our motors or create things like overshoot and oscillations, resulting in unstable results. The development to centralized control with inverse dynamics resulted in movements that still had the snappiness required to get to the target but were much smoother in their approach, however some jittering after catching the ball remained present. The final development to Robust control really helped to reduce our final state jitter, as well as provide the highest catch rate.

From the performance of each controller, they could be ranked in order of decreasing performance overall as shown: Robust Joint Space Control, Decentralized PID Joint Control, Joint Space Inverse Dynamics Control, and Task Space Inverse Dynamics Control. The Joint Space Robust controller was able to achieve optimal responsiveness with smoothest movements of the controllers, which facilitated secure capture of the ball along its trajectory. While there were cases of oscillation in motion and position maintenance, this had minimal impact on the overall catching of the ball. The Decentralized PID controller was placed as the next highest performance because it enabled a higher response time, with some jerkiness within the movements of the arm. The sudden changes in velocity from the jerk in the arm movement had minimal impact on the overall movement to the ball location, but this also made the success rate sensitive to slight inaccuracies in achieving the final location of the ball trajectory. If the ball reached the a location that would hit the edge of the hoop in the arm itself with the Decentralized PID control system, the jerkiness in the ball movement would sometimes hit the ball out of the hoop when trying to land inside. An instance of this can be seen with figure 11, where the fast motion of the ball would make positioning of the arm's hoop sensitive to excessive jerkiness in the motion. The overall performance of the inverse dynamics controllers were similar, but the performance of the inverse dynamics controller in the joint space was slightly better because it seemed to be less prone to the erratic movement to the side discussed within the results section. This may potentially be a result of the more computations of the forward and inverse kinematics necessary for the task space inverse dynamics controller, where enforcing limits and error handling in the controller was potentially easier through the control in joint space.

However, when considering the overall control efficiency of the systems tested, the following could be the ranking of the controllers tested in decreasing order of efficiency: Robust Joint Space Control, Joint Space Inverse Dynamics Control, Task Space Inverse Dynamics Control, Decentralized PID Joint Control. As a disclaimer, the following discussion on the controller efficiency in performance is mainly qualitative, as obtaining more quantitative measurements of error values wasn't feasible with the time constraints placed on the project timeline. The Robust Joint Space controller seemed to demonstrate the best efficiency in operation with the smoother movements and minimal jitter, indicating the capture to be achieved with minimal saturation of the velocity values during movement. This was captured in figure 17, where the ball capture was successful despite the tracking of the ball being close to the edge of the hoop. Excluding cases of fatal errors from erratic movement discussed in the results section, both inverse dynamics controllers demonstrated similar amounts of efficiency during performance from minimal amounts of jerky movements to the end positions. This is due to the movements of the system accounting for dynamics in the centralized control scheme, where the motion was more controlled. The operational space joint controller was ranked lower in terms of efficiency because of there being more overshoot and jitter present during iterations of the testing processes. Finally, the Decentralized PID controller was placed last because it seemed to have the most jerk in its movements to the goal positions. As compared to the centralized controllers, the independent control of the joints allowed for application of higher joint velocities disregarding the impact they may have on the system dynamics. This potentially increases the necessary power needed from the actuators to counteract the impact of the motion from other joints in moving/maintaining a position or velocity desired.

Limitations within the performance of this project's system arose with problems related to latency and hardware limitations of the system. Actuator joint velocity limits were a component that hindered the overall performance potentially. These values set limits on the tuning process of the developed controllers, limiting the increase of gains for faster responsiveness of the system. This was potentially seen with the tuning of the inverse dynamics and robust controllers. When raising the proportional gain values of the controller components, there was an observed maximum amount of responsiveness achieved

within the operation of the system in its motion to the ball itself. Increasing these gain values further made the controller motion exhibit more cases of overshoot and jittering that weren't beneficial to the ball catching motion. This was also present in the decentralized PID controller, where convergence to the final position value was more difficult to obtain because of the sensitivity in tuning the other gain values for the joints. Thus, the tuning of higher gain values was deemed not possible within the constraints of this project.

The lag present in the controller's response to the ball catching was also likely caused with the limitations of the camera. With the current iteration of the project, the camera had a rate of 120fps, where the target location was generated from a prediction of the trajectory intersection point through computation of the height and speed values. Within the process, latency potentially arises in the processing of the ball image in the beginning. In addition, separate filters were applied to the resulting centroid and approximate height computed to generate a final coordinate goal to send to the controller. As the system utilizes C++ for execution of this script, the overall processing speed costs from these computations would ideally be minimal, but further development of the algorithm for optimal run time would potentially be beneficial to reducing the overall latency of the system. Another potential cause for drops in performance could arise from discrepancies in the overall operation frequency of the system. Retrieving the operational frequency using the ROS2 functions would sometimes show dips in the operational frequency from the ideal value of 500Hz. From the hardware side, a higher frame rate for the camera would potentially reduce this further, and investigation of depth-sensing cameras would also potentially benefit this process overall.

Apart from latency, another issue that was sometimes evident in the system performance was a slight inaccuracy in meeting the end-effector location goal. Causes of this potentially arise from the resolution of the hardware systems, and the overall computed values from the controllers. The overall size of the image of the workspace captured in the workspace was 320x240 pixels. As the ball bounce also rose to significant heights relative to the position of the camera, the field of view of the camera also had some impact on the overall processing and prediction of the ball position. Within the operation of the full system, the issue with processing the image of the ball could also be seen as the arm showed different behaviors in responsiveness according to different heights the ball was manually moved within. When identifying errors from the computation of the controller itself, potential causes could arise from the modeled dynamics of the system. The inertia of the link components were computed through an approximation of the physical parameters that were 3D printed, which potentially introduced significant variations in the location of center of mass in the physical component.

VII. Future Directions

As mentioned within the discussion, the next step within the implementation and analysis of the system would potentially include a method for quantifying the efficiency of the controller performance in the system. The efficiency was qualitatively defined with the observed jerkiness in the arm motion, but utilizing the actuator's data collection capabilities would introduce more credibility behind this assumption. Within this process, the effort values of the controller could potentially be measured via intaken values for current and speed, as they would be proportional in approximating values of power spent within the system. Obtaining a cumulative value of power during a step input or catching motion would give more insight to quantitative values in the controller efficiency relative to each other.

Within steps for improving the performance of the system, the investigations into hardware performance mentioned in the discussion would likely be beneficial in optimizing the responsiveness and accuracy of the arm itself. Further selection of the camera in particular to minimize the noise of the image processing would likely significantly improve the ball location predictions made from the detection system. With the current ROS2 implementation being set to achieve operation of 500Hz, obtaining camera systems and actuators that would exploit this rate of communication would bring the overall system closer to a theoretical max performance.

In further analysis of the potential theoretical maximum performance of the system, a potential real-sim-real workflow could be implemented for future project implementations. As latency seemed to be a significant issue with system performance, conducting implementation and gain tuning in simulation would give insight to the max performance (when applying the necessary joint limits). Within this project, an implementation of a simulation mimicking the implementation of Decentralized PID control was done to further analyze this parameter. This was done integrating the ROS2 framework of the project into the Ignition Gazebo Fortress simulation software, and the developed simulation environment can be seen with figure 18.

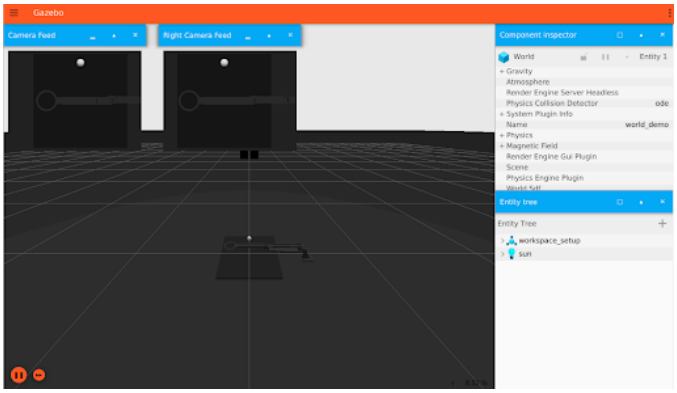


Fig. 18: Image of Simulation Setup

Full exploration of the gain tuning wasn't able to be conducted due to time constraints on the project, but future developments of this project could include integration of all the controllers tested within the hardware component of

this project. Exploring tuning of the simulated gains for the controllers would then give more comprehensive insight into the overall performance of the systems comparatively, and these results could then be compared with the hardware implementation. This would help to further isolate issues with the system's hardware implementation and give more information about potential actions to minimize these problems.

VIII. Appendix

A. Custom CAD Design

Figure 19 shows the mechancial assembly of the robot catcher system made through Onshape.

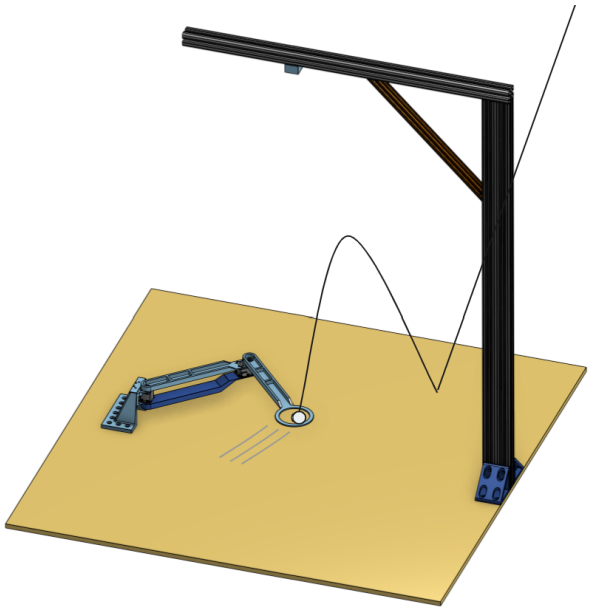


Fig. 19: CAD of System Mechanical Setup

B. Team Members

- Manuel Gonzalez Parra: Computer Vision, Independent Joint Controller Design
- Justin Lee: System Architecture, PWM control setup, Inverse Dynamics Controller Design
- Trevor Oshiro: Actuator Setup, Mechanical Design, Robust Controller Design, Final Controller Testing
- Eli Whitaker: Experimental Setup, Operational Space Inverse Dynamics Controller Design

C. System Block Diagram

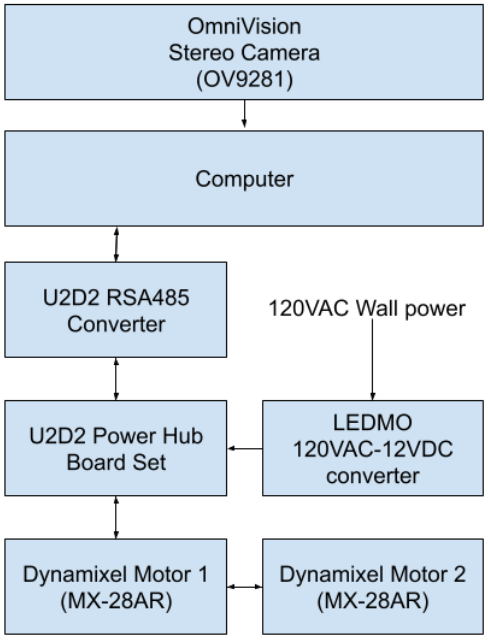


Fig. 20: System Block Diagram

D. Bill of Materials

The materials used for this project can be found in Table I.

TABLE I: Bill of Materials

Component	Description	QTY	Unit Price	Vendor
Dynamixel MX-28AR	All-in-one smart actuator	2	$329.89	ROBOTIS
HN07-N101 Horn Set	Horn bearing	2	$11.66	ROBOTIS
HN07-I101 Idler Set	Idler bearing	2	$18.59	ROBOTIS
Robot Cable-4P	4-pin DYNAMIXEL cable	3	$22.22 (10 pk)	ROBOTIS
U2D2 Power Hub Board Set	Power hub for DYNAMIXEL	1	$20.90	ROBOTIS
LEDMO Power Supply	12 V / 5 A desktop PSU	1	$11.99	Amazon
U2D2	USB-to-RS-485 converter	1	$35.31	ROBOTIS
Lenovo ThinkPad P1	Mobile workstation	1	N/A	Lenovo
OmniVision OV9281 Stereo Cam	High-speed camera module	1	$88.99	Arducam
M2 × 0.4 mm SHCS	Screws for horns / idlers	36	N/A	McMaster-Carr
M2.5 × 0.45 mm SHCS	Screws for actuator mounts	24	N/A	McMaster-Carr
USB-C → USB-A Cable	Data / power cable	1	N/A	Amazon
USB-A → Micro-USB Cable	Data / power cable	1	N/A	Amazon
Steel Panel	2 ft × 3 ft base plate	1	N/A	Home Depot
8020 Rails	T-slot aluminum extrusion	2	N/A	8020 Inc.
8020 Brackets	T-slot brackets	3	N/A	8020 Inc.
¼-20 Bolts	Bolts for camera mount	36	N/A	Home Depot
Arm Base	3-D printed PLA base	1	N/A	
Arm Link Pieces	3-D printed PLA links	3	N/A	
Net	Black mesh catching net	1	$4.99	Amazon
Ping-Pong Balls	Test projectiles (12-pk)	12	$5.99	Amazon
Black Matte Paint	System finish spray paint	1	$3.99	Amazon / Home Depot
Black Felt Fabric	System finish cloth	1	$8.99	Amazon

E. Reference Code

The Github repository for this project can be found with this link: https://github.com/rovertio/MAE263C_CatcherProject

IX. Bibliography

References

[1] Y. Zhang, T. Liang, Z. Chen, Y. Ze, and H. Xu, "Catch it! learning to catch in flight with mobile dexterous hands," in *Proc. IEEE Int. Conf. Robot. Autom.*, Atlanta, GA, USA, May 2025.

[2] A. Dastider, H. Fang, and M. Lin, "Unified control framework for real-time interception and obstacle avoidance of fast-moving objects with diffusion variational autoencoder," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Abu Dhabi, UAE, Oct. 2024, pp. 13883–13890.

[3] B. Bäuml, T. Wimböck, and G. Hirzinger, "Kinematically optimal catching a flying ball with a hand-arm system," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Taipei, Taiwan, Oct. 2010, pp. 2592–2599.

[4] S. S. M. Salehian, M. Khoramshahi, and A. Billard, "A dynamical system approach for catching softly a flying object: theory and experiment," *IEEE Trans. Robot.*, vol. 32, no. 3, pp. 462–471, Jun. 2016.

[5] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: modelling, planning and control*, London, U.K.: Springer, 2009.
[6] OpenAI, "ChatGPT: best practices in ROS 2 and C++," Tech. Rep., May 2025.
[7] ROBOTIS Co., Ltd., "Dynamixel Workbench for ROS API," e-Manual, rev. 1.45, 2024. [Online]. Available: https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_workbench/. Accessed: 14-Jun-2025.
[8] OpenCV Team, "Morphological transformations," *OpenCV Documentation*, 2024. [Online]. Available: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html. Accessed: 14-Jun-2025.
[9] S. Suzuki and K. Abe, "Topological structural analysis of digitized binary images by border following," *Comput. Vision Graph. Image Process.*, vol. 30, no. 1, pp. 32–46, Jan. 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0734189X85900167. Accessed: 14-Jun-2025.
[10] OpenCV Team, "Image moments," *OpenCV Documentation*, 2024. [Online]. Available: https://docs.opencv.org/3.4/d0/d49/tutorial_moments.html. Accessed: 14-Jun-2025.
[11] Columbia University Computer Vision, "Pinhole camera and stereo vision lecture notes," Autonomous Mobile Robots Course, Columbia Univ., 2017. [Online]. Available: https://www.cs.columbia.edu/~allen/F17/NOTES/Lecture_5_rev_3.pdf. Accessed: 14-Jun-2025.

Ball Predictor Node

```
/* ========================================================================
 * ball_predictor_node.cpp – stereo blob tracker + parabolic landing
 * ------------------------------------------------------------------------
 *  • subscribes   /camera/image_raw   (640×240 stereo image)
 *  • publishes    /ball_centroid      (geometry_msgs/Point, px coords)
 * ======================================================================== */
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/image.hpp>
#include <geometry_msgs/msg/point.hpp>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/imgproc.hpp>
#include <deque>
#include <cmath>
#include <algorithm>

using std::placeholders::_1;

/* ── shared vision constants (duplicate of detector.cpp – factor out later) */
namespace vp {
constexpr int    IMG_W = 640, IMG_H = 240;
constexpr int    HALF_W = IMG_W / 2;
constexpr double IMG_CX = HALF_W / 2.0;
constexpr double IMG_CY = IMG_H  / 2.0;

constexpr int    THRESH    = 225;
constexpr int    ERODE_SZ  = 3, DILATE_SZ = 7;
constexpr double MIN_AREA  = 15.0 * 15.0;

constexpr double HFOV_DEG  = 45.0;
constexpr double CAMERA_H  = 85.0;          // camera height above table [cm]
constexpr double BASELINE  = 5.5;           // stereo baseline [cm]
constexpr double FOCAL_PX  = (HALF_W/2.0) /
        std::tan((HFOV_DEG*M_PI/180.0)/2.0);

/* Physics & filter */
constexpr double G_CM      = 981.0;         // gravity [cm·s⁻²]
constexpr double LPF_ALPHA = 0.8;           // pos low-pass
constexpr double VEL_ALPHA = 0.8;           // vel low-pass
constexpr double PRED_ALPHA= 0.8;           // landing-point LPF
constexpr size_t  BUF_MAX  = 4;
constexpr size_t  BUF_MIN  = 2;
constexpr double  RESET_GAP_S = 0.30;
} // namespace vp

/* ---- centroid helper (identical to detector.cpp) ------------------ */
static bool centroid(const cv::Mat& roi,
                     double& cx,double& cy,double& area,
                     const cv::Mat& erodeK,const cv::Mat& dilateK)
{
  cv::Mat g,m; cv::cvtColor(roi,g,cv::COLOR_BGR2GRAY);
  cv::threshold(g,m,vp::THRESH,255,cv::THRESH_BINARY);
  cv::erode(m,m,erodeK); cv::dilate(m,m,dilateK);
```

```
  std::vector<std::vector<cv::Point>> cont;
  cv::findContours(m,cont,cv::RETR_EXTERNAL,cv::CHAIN_APPROX_SIMPLE);
  if(cont.empty()) return false;
  auto it = std::max_element(cont.begin(),cont.end(),
        [](auto&a,auto&b){ return cv::contourArea(a)<cv::contourArea(b); });
  area = cv::contourArea(*it);
  if(area < vp::MIN_AREA) return false;
  cv::Moments mom = cv::moments(*it);
  cx = mom.m10 / mom.m00;
  cy = mom.m01 / mom.m00;
  return true;
}

/* ================================================================== */
class BallPredictorNode : public rclcpp::Node
{
public:
  BallPredictorNode():Node("ball_predictor_node")
  {
    img_sub_ = create_subscription<sensor_msgs::msg::Image>(
        "/camera/image_raw",10,std::bind(&BallPredictorNode::cb_img,this,_1));
    pub_=create_publisher<geometry_msgs::msg::Point>("/ball_centroid",10);

    erodeK_=cv::getStructuringElement(cv::MORPH_ELLIPSE,{vp::ERODE_SZ,vp::ERODE_SZ});

dilateK_=cv::getStructuringElement(cv::MORPH_ELLIPSE,{vp::DILATE_SZ,vp::DILATE_SZ});
    RCLCPP_INFO(get_logger(),"Predictor ready (g=%.1f cm/s²)",vp::G_CM);
  }

private:
/* ---- helper: compute height from area & disparity ----------------- */
  double height_cm(double area,double disparity) const
  {
    const double AREA_REF = 340.0, H_SCALE = 120.0;
    double h_area = std::max(0.0, H_SCALE*(std::sqrt(AREA_REF/area)-1.0));
    if(std::abs(disparity) < 1e-2) return h_area; // avoid div-by-zero
    double dist = (vp::BASELINE*vp::FOCAL_PX)/disparity;
    double h_st = vp::CAMERA_H - dist;
    return 0.6*h_st + 0.4*h_area;               // cheap fuse
  }

/* ---- main callback ------------------------------------------------ */
  void cb_img(const sensor_msgs::msg::Image::ConstSharedPtr& msg)
  {
    /* ··· basic checks ··· */
    auto cv_ptr = cv_bridge::toCvCopy(msg,sensor_msgs::image_encodings::BGR8);
    const cv::Mat& img = cv_ptr->image;
    if(img.empty()) return;
    if(img.cols!=vp::IMG_W || img.rows!=vp::IMG_H){
      RCLCPP_WARN_THROTTLE(get_logger(),*get_clock(),5000,
                           "Unexpected image size %dx%d",img.cols,img.rows);
      return;
    }

    /* ··· extract centroids from left/right eye ··· */
```

```
cv::Rect roiL(0,0,vp::HALF_W,vp::IMG_H), roiR(vp::HALF_W,0,vp::HALF_W,vp::IMG_H);
double lx,ly,aL=0; bool fL = centroid(img(roiL),lx,ly,aL,erodeK_,dilateK_);
double rx,ry,aR=0; bool fR = centroid(img(roiR),rx,ry,aR,erodeK_,dilateK_);

if(!(fL||fR)){   // nothing
  if((get_clock()->now()-last_seen_).seconds()>vp::RESET_GAP_S){
    buf_.clear(); has_prev_h_=has_prev_v_=has_pred_=false;
  }
  return;
}
last_seen_ = get_clock()->now();

/* choose centroid */
double px,py,area;
if(fL&&fR){ px=(lx+rx)/2; py=(ly+ry)/2; area=(aL+aR)/2; }
else if(fL){ px=lx; py=ly; area=aL; }
else        { px=rx; py=ry; area=aR; }

double disp = (fL&&fR)? lx-rx : 0.0;

/* height LPF */
double h_raw = height_cm(area,disp);
if(has_prev_h_) h_raw = vp::LPF_ALPHA*prev_h_ + (1.0-vp::LPF_ALPHA)*h_raw;
prev_h_=h_raw; has_prev_h_=true;

/* perspective correct */
double corr = (vp::CAMERA_H-h_raw)/vp::CAMERA_H;
px = vp::IMG_CX + (px-vp::IMG_CX)*corr;
py = vp::IMG_CY + (py-vp::IMG_CY)*corr;

/* buffer sample */
if(buf_.size()==vp::BUF_MAX) buf_.pop_front();
buf_.push_back({px,py,h_raw,get_clock()->now().seconds()});

if(buf_.size()<vp::BUF_MIN) return;

/* planar velocity */
const auto& q0=buf_[buf_.size()-2];
const auto& q1=buf_.back();
double dt = q1.t - q0.t;
if(dt<1e-3) return;      // avoid div0
double vx=(q1.x - q0.x)/dt, vy=(q1.y - q0.y)/dt;
if(has_prev_v_){
  vx = vp::VEL_ALPHA*prev_vx_ + (1.0-vp::VEL_ALPHA)*vx;
  vy = vp::VEL_ALPHA*prev_vy_ + (1.0-vp::VEL_ALPHA)*vy;
}
prev_vx_=vx; prev_vy_=vy; has_prev_v_=true;

/* time until ground (vertical parabola) */
double vz   = (q1.h - q0.h) / dt;
double h_cl = std::max(0.0, q1.h);              // ← 1. clamp height
double disc = vz*vz + 2*vp::G_CM*h_cl;          // ← 2. safe radicand
if (disc <= 0.0) return;                        // ← 3. give up this frame

double t_land = (-vz - std::sqrt(disc)) / -vp::G_CM;
```

```
    double px_land = px + vx*t_land;
    double py_land = py + vy*t_land;

    if(has_pred_){
      px_land = vp::PRED_ALPHA*prev_pred_x_ + (1.0-vp::PRED_ALPHA)*px_land;
      py_land = vp::PRED_ALPHA*prev_pred_y_ + (1.0-vp::PRED_ALPHA)*py_land;
    }
    prev_pred_x_=px_land; prev_pred_y_=py_land; has_pred_=true;

    px_land = std::clamp(px_land, 0.0, static_cast<double>(vp::HALF_W  - 1)); // 0 …
319
    py_land = std::clamp(py_land, 0.0, static_cast<double>(vp::IMG_H   - 1)); // 0 …
239


    /* publish */
    geometry_msgs::msg::Point out;
    out.x=px_land; out.y=py_land; out.z=h_raw;
    pub_->publish(out);
  }

/* ---- sample struct & state ------------------------------------- */
  struct Sample{ double x,y,h,t; };
  std::deque<Sample> buf_;

  rclcpp::Time last_seen_{0,0,RCL_ROS_TIME};
  double prev_h_{0}; bool has_prev_h_{false};
  double prev_vx_{0}, prev_vy_{0}; bool has_prev_v_{false};
  double prev_pred_x_{0}, prev_pred_y_{0}; bool has_pred_{false};

  rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr img_sub_;
  rclcpp::Publisher<geometry_msgs::msg::Point>::SharedPtr pub_;
  cv::Mat erodeK_, dilateK_;
};

/* ---- main ------------------------------------------------------ */
int main(int argc,char** argv)
{
  rclcpp::init(argc,argv);
  rclcpp::spin(std::make_shared<BallPredictorNode>());
  rclcpp::shutdown();
  return 0;
}
```

Pixel to XY Bridge

```
#include <memory>
#include <functional>
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/point.hpp"
#include "controller_msgs/msg/set_xy.hpp"

using SetXY = controller_msgs::msg::SetXY;      // alias once

class PixelBridge : public rclcpp::Node
{
public:
  PixelBridge() : Node("pixel_to_xy_bridge")
  {
    using std::placeholders::_1;
    sub_ = create_subscription<geometry_msgs::msg::Point>(
              "/ball_centroid", 10, std::bind(&PixelBridge::cb, this, _1));

    pub_ = create_publisher<SetXY>("/set_xy", 10);   // ★ changed
  }

private:
  void cb(const geometry_msgs::msg::Point::SharedPtr p)
  {
    constexpr double SCALE = 0.22;
    constexpr double PIX_H = 240.0;
    constexpr double PIX_W = 320.0;

    double dx = (PIX_H / 2.0) - p->y;
    double dy = (PIX_W / 2.0) - p->x;

    SetXY msg;                                       // ★ changed
    msg.x = dx * SCALE;
    msg.y = dy * SCALE;
    pub_->publish(msg);
  }

  rclcpp::Subscription<geometry_msgs::msg::Point>::SharedPtr sub_;
  rclcpp::Publisher<SetXY>::SharedPtr                        pub_;  // ★ changed
};

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PixelBridge>());
  rclcpp::shutdown();
  return 0;
}
```

Independent Joint Controller

```
/* ====================================================================
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *  Computes     :  planar 2-DOF Independent PID Joint Control ➜ pwm signal
 *  Writes to    :  Dynamixel PWM Registers
 *  Publishes on :  /pwm/joint_plot_data  (controller_msgs/JointPlotData)
 * ==================================================================== */

#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"
#include <cmath>

using SetXY = controller_msgs::msg::SetXY;
using Plot = controller_msgs::msg::JointPlotData;
using std::placeholders::_1;

class PwmPositionNode : public rclcpp::Node
{
public:
  PwmPositionNode() : Node("pwm_position_node")
  {
    declare_parameter<std::string>("port", "/dev/ttyUSB0");
    declare_parameter<int>("baud", 1000000);
    declare_parameter<double>("kp", 50.0);
    declare_parameter<double>("ki", 0.1);
    declare_parameter<double>("kd", 5.0);
    declare_parameter<int>("max_pwm", 885);
    declare_parameter<int>("ctrl_rate_hz", 1000);

    kp_ = get_parameter("kp").as_double();
    ki_ = get_parameter("ki").as_double();
    kd_ = get_parameter("kd").as_double();
    max_pwm_ = get_parameter("max_pwm").as_int();
    dt_s_ = 1.0 / get_parameter("ctrl_rate_hz").as_int();

    port_ =
dynamixel::PortHandler::getPortHandler(get_parameter("port").as_string().c_str());
    packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
    if (!port_->openPort() || !port_->setBaudRate(get_parameter("baud").as_int()))
      RCLCPP_FATAL(get_logger(), "Serial open failed");

    for (uint8_t id : control::SERVO_IDS)
    {
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
      write8(id, control::ADDR_OPERATING_MODE, control::MODE_PWM);
      write8(id, control::ADDR_TORQUE_ENABLE, 1);
    }

    sub_xy_ = create_subscription<SetXY>("set_xy", 10,
std::bind(&PwmPositionNode::cb_xy, this, _1));
    pub_plot_ = create_publisher<Plot>("pwm/joint_plot_data", 10);
```

```
    timer_ = create_wall_timer(std::chrono::duration<double>(dt_s_),
std::bind(&PwmPositionNode::control_loop, this));
    last_good_cmd_time_ = now();

    RCLCPP_INFO(get_logger(), "PWM node ready (kp=%.1f ki=%.1f kd=%.1f)", kp_, ki_,
kd_);
  }

  ~PwmPositionNode() override
  {
    for (uint8_t id : control::SERVO_IDS)
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
    port_->closePort();
  }

private:
  void write8(uint8_t id, uint16_t addr, uint8_t d)
  {
    uint8_t e{};
    int rc = packet_->write1ByteTxRx(port_, id, addr, d, &e);
    if (rc != COMM_SUCCESS || e)
      RCLCPP_ERROR_THROTTLE(get_logger(), *get_clock(), 2000, "DXL %u err=%d rc=%d",
id, e, rc);
  }

  // Use GroupSyncWrite to set both motors' PWMs at once
  void send_pwms(int pwm1, int pwm2)
  {
    dynamixel::GroupSyncWrite group_writer(port_, packet_, control::ADDR_GOAL_PWM, 2);
    uint8_t b1[2] = {DXL_LOBYTE(pwm1), DXL_HIBYTE(pwm1)};
    uint8_t b2[2] = {DXL_LOBYTE(pwm2), DXL_HIBYTE(pwm2)};
    bool ok1 = group_writer.addParam(control::ID1, b1);
    bool ok2 = group_writer.addParam(control::ID2, b2);
    if (!ok1 || !ok2)
    {
      RCLCPP_ERROR_THROTTLE(get_logger(), *get_clock(), 2000,
                            "GroupSyncWrite addParam failed! (ok1=%d ok2=%d)", ok1,
ok2);
      return;
    }
    int rc = group_writer.txPacket();
    if (rc != COMM_SUCCESS)
    {
      RCLCPP_ERROR_THROTTLE(get_logger(), *get_clock(), 2000,
                            "GroupSyncWrite PWM failed: %s",
packet_->getTxRxResult(rc));
    }
  }

  void cb_xy(const SetXY::SharedPtr msg)
  {
    if (!std::isfinite(msg->x) || !std::isfinite(msg->y))
    {
      RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000, "set_xy contains NaN/Inf
— ignoring");
```

```
      return;
    }
    double j1, j2;
    RCLCPP_INFO(get_logger(), "PWM HERE->>>>> (kp=%.1f ki=%.1f kd=%.1f)", kp_, ki_,
kd_);
    if (control::ik_xy(msg->x * 0.01, msg->y * 0.01, j1, j2))
    {
      tgt_j1_ = j1 * 180.0 / M_PI;
      tgt_j2_ = j2 * 180.0 / M_PI;
      last_good_cmd_time_ = now();
      has_target_ = true;
    }
    else
    {
      RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000, "IK failed for (%.1f,
%.1f)", msg->x, msg->y);
    }
  }

  void control_loop()
  {
    if (!has_target_ || (now() - last_good_cmd_time_).seconds() > 3.0)
      return;
    int32_t tick1{}, tick2{};
    control::read_two_positions(port_, packet_, tick1, tick2);
    double cur1 = control::tick2deg(0, tick1);
    double cur2 = control::tick2deg(1, tick2);
    RCLCPP_INFO(get_logger(), "PWM gains (kp=%.1f ki=%.1f kd=%.1f)", kp_, ki_, kd_);
    double pwm1 = pid1_.step(tgt_j1_ - cur1, kp_, ki_, kd_, max_pwm_);
    double pwm2 = pid2_.step(tgt_j2_ - cur2, kp_, ki_, kd_, max_pwm_);
    send_pwms(int(pwm1), int(pwm2));

    Plot p;
    p.stamp = now();
    p.q1_deg = cur1;
    p.q2_deg = cur2;
    p.q1_des_deg = tgt_j1_;
    p.q2_des_deg = tgt_j2_;
    p.e1 = tgt_j1_ - cur1;
    p.e2 = tgt_j2_ - cur2;
    p.pwm1 = pwm1;
    p.pwm2 = pwm2;
    p.measured_q1 = cur1;
    p.measured_q2 = cur2;
    p.desired_q1 = tgt_j1_;
    p.desired_q2 = tgt_j2_;
    p.controller_name = "pwm_position_node";
    pub_plot_->publish(p);
  }

  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
  rclcpp::Publisher<Plot>::SharedPtr pub_plot_;
  rclcpp::TimerBase::SharedPtr timer_;
  dynamixel::PortHandler *port_{};
  dynamixel::PacketHandler *packet_{};
```

```
  control::PidState pid1_, pid2_;
  double kp_, ki_, kd_, dt_s_;
  int max_pwm_;
  bool has_target_{false};
  double tgt_j1_{}, tgt_j2_{};
  rclcpp::Time last_good_cmd_time_;
};

/* ---- main ---------------------------------------------------------- */
int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PwmPositionNode>());
  rclcpp::shutdown();
  return 0;
}
```

Joint Space Inverse Dynamics Controller

```
/* ============================================================================
 * joint_space_idc.cpp — latency-optimised   +   joint-limit clamp
 * ----------------------------------------------------------------------------
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *                  ↳ target (x,y) is converted to desired joint angles via IK
 *  Computes     :  planar 2-DOF **Joint-Space** Inverse Dynamics Control
 *                  τ = B(q) q̈* + n(q, q̇)  with  q̈* = Kp (qd-q) + Kd (-q̇)
 *  Writes to    :  Dynamixel Goal-Position registers (position mode)
 *  Publishes on :  /idc_joint/joint_plot_data  (controller_msgs/JointPlotData)
 *
 *  Style & low-level helpers follow the operational-space IDC node
:contentReference[oaicite:0]{index=0}
 * ============================================================================ */
#include <chrono>
#include <cmath>
#include <algorithm>
#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include <Eigen/Dense>

#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"

#ifndef PROFILING
#define PROFILING 1
#endif

using SetXY = controller_msgs::msg::SetXY;
using Plot  = controller_msgs::msg::JointPlotData;
namespace ch = std::chrono;
using namespace std::chrono_literals;
using std::placeholders::_1;

/* ── Joint hard-limits (deg) ─────────────────────────────────────────── */
constexpr double Q1_MIN_DEG = -90.0;
constexpr double Q1_MAX_DEG =  20.0;
constexpr double Q2_MIN_DEG =  10.0;
constexpr double Q2_MAX_DEG = 140.0;

/* ═══════════════════════════════════════════════════════════════════════ */
class JointSpaceIDCNode final : public rclcpp::Node
{
public:
  JointSpaceIDCNode() : Node("joint_space_idc")
  {
    /* ── parameters ──────────────────────────────────────────────────── */
    declare_parameter("port",           std::string{"/dev/ttyUSB0"});
    declare_parameter("baud",           1'000'000);
    declare_parameter("kp1_joint",      50.0);
    declare_parameter("kd1_joint",       1.0);
```

```
  declare_parameter("kp2_joint",      50.0);
  declare_parameter("kd2_joint",       1.0);
  declare_parameter("ctrl_rate_hz",   500);

  kp1_ = get_parameter("kp1_joint").as_double();
  kd1_ = get_parameter("kd1_joint").as_double();
  kp2_ = get_parameter("kp2_joint").as_double();
  kd2_ = get_parameter("kd2_joint").as_double();
  dt_  = 1.0 / get_parameter("ctrl_rate_hz").as_int();

  /* ── Dynamixel initialisation ──────────────────────────────── */
  port_   = dynamixel::PortHandler::getPortHandler(
              get_parameter("port").as_string().c_str());
  packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
  if (!port_->openPort() ||
      !port_->setBaudRate(get_parameter("baud").as_int()))
    throw std::runtime_error("DXL: cannot open port or set baud");

  /* bulk read   : 8 bytes / ID  (vel + pos) */
  bulk_ = new dynamixel::GroupBulkRead(port_, packet_);
  sync_ = new dynamixel::GroupSyncWrite(port_, packet_,
                                        control::ADDR_GOAL_POSITION, 4);
  constexpr uint16_t VEL_ADDR = control::ADDR_PRESENT_VELOCITY;
  if (!bulk_->addParam(control::ID1, VEL_ADDR, 8) ||
      !bulk_->addParam(control::ID2, VEL_ADDR, 8))
    throw std::runtime_error("DXL: GroupBulkRead addParam failed");

  for (uint8_t id : control::SERVO_IDS) {
    write8(id, control::ADDR_TORQUE_ENABLE, 0);
    write8(id, control::ADDR_OPERATING_MODE, control::MODE_POSITION);
    write8(id, control::ADDR_TORQUE_ENABLE, 1);
  }

  /* ── ROS 2 I/O ─────────────────────────────────────────────── */
  sub_xy_   = create_subscription<SetXY>(
                "set_xy", 10, std::bind(&JointSpaceIDCNode::cb_target, this, _1));
  pub_plot_ = create_publisher<Plot>("idc_joint/joint_plot_data", 10);
  timer_    = create_wall_timer(
                ch::duration<double>(dt_),
                std::bind(&JointSpaceIDCNode::cb_loop, this));

  RCLCPP_INFO(get_logger(),
    "Joint-Space IDC ready | dt=%.4f s (%.0f Hz)  kp=[%.1f,%.1f] kd=[%.1f,%.1f]",
    dt_, 1.0/dt_, kp1_, kp2_, kd1_, kd2_);
}

~JointSpaceIDCNode() override
{
  for (uint8_t id : control::SERVO_IDS)
    write8(id, control::ADDR_TORQUE_ENABLE, 0);
  delete bulk_;
  delete sync_;
  port_->closePort();
}
```

```
private:
  /* ── low-level helpers ──────────────────────────────────────────── */
  void write8(uint8_t id, uint16_t addr, uint8_t data)
  { uint8_t err{}; packet_->write1ByteTxRx(port_, id, addr, data, &err); }

  bool read_states(double &q1, double &q2, double &dq1, double &dq2)
  {
    if (bulk_->txRxPacket() != COMM_SUCCESS) return false;
    auto ready = [&](uint8_t id, uint16_t a){ return bulk_->isAvailable(id, a, 4); };
    if (!(ready(control::ID1, control::ADDR_PRESENT_POSITION) &&
          ready(control::ID1, control::ADDR_PRESENT_VELOCITY) &&
          ready(control::ID2, control::ADDR_PRESENT_POSITION) &&
          ready(control::ID2, control::ADDR_PRESENT_VELOCITY)))
      return false;

    int32_t v1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t p1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_POSITION, 4);
    int32_t v2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t p2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_POSITION, 4);

    dq1 = control::vel_tick2rad(v1);
    dq2 = control::vel_tick2rad(v2);
    q1  = control::tick2deg(0, p1) * M_PI/180.0;
    q2  = control::tick2deg(1, p2) * M_PI/180.0;
    return true;
  }

  void send_goal(uint32_t t1, uint32_t t2)
  {
    uint8_t b1[4] = { DXL_LOBYTE(DXL_LOWORD(t1)), DXL_HIBYTE(DXL_LOWORD(t1)),
                      DXL_LOBYTE(DXL_HIWORD(t1)), DXL_HIBYTE(DXL_HIWORD(t1)) };
    uint8_t b2[4] = { DXL_LOBYTE(DXL_LOWORD(t2)), DXL_HIBYTE(DXL_LOWORD(t2)),
                      DXL_LOBYTE(DXL_HIWORD(t2)), DXL_HIBYTE(DXL_HIWORD(t2)) };
    sync_->clearParam();
    sync_->addParam(control::ID1, b1);
    sync_->addParam(control::ID2, b2);
    sync_->txPacket();
  }

#ifdef PROFILING
  struct Stat { double avg=0, worst=0;
    void update(double s){ const double k=.01; avg = (1-k)*avg + k*s; worst =
std::max(worst, s); } } st_[5];
  uint64_t loops_{0};
#endif

  /* ── callbacks ──────────────────────────────────────────────────── */
  void cb_target(const SetXY::SharedPtr msg)
  {
    if (!std::isfinite(msg->x) || !std::isfinite(msg->y)) return;

    double j1, j2;
    if (!control::ik_xy(msg->x * 0.01, msg->y * 0.01, j1, j2)) {
      RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000,
                           "IK failed for (%.1f, %.1f) cm", msg->x, msg->y);
```

```
    return;
  }
  qd_ = { j1, j2 };
  has_target_ = true;
}

void cb_loop()
{
  auto t0 = ch::steady_clock::now();
  if (!has_target_) return;

  /* 1 ▸ state read ------------------------------------------------ */
  double q1, q2, dq1, dq2;
  if (!read_states(q1, q2, dq1, dq2)) return;
  auto t1 = ch::steady_clock::now();

  Eigen::Vector2d q{ q1, q2 }, dq{ dq1, dq2 };

  /* 2 ▸ inverse dynamics ------------------------------------------ */
  Eigen::Vector2d e  = qd_ - q;
  Eigen::Vector2d ed = -dq;                    // desired qd̈ = 0
  Eigen::Vector2d qdd{ kp1_*e[0] + kd1_*ed[0],
                       kp2_*e[1] + kd2_*ed[1] };

  Eigen::Vector2d tau = control::mass_matrix(q) * qdd +
                        control::coriolis(q, dq) +
                        control::gravity(q);
  auto t2 = ch::steady_clock::now();

  /* 3 ▸ convert τ → Δθdeg and command ----------------------------- */
  Eigen::Vector2d dqdeg = control::tau_to_deg(tau);
  Eigen::Vector2d qdeg  = (q * 180.0 / M_PI) + dqdeg;

  qdeg[0] = std::clamp(qdeg[0], Q1_MIN_DEG, Q1_MAX_DEG);
  qdeg[1] = std::clamp(qdeg[1], Q2_MIN_DEG, Q2_MAX_DEG);

  send_goal(control::deg2tick(0, qdeg[0]),
            control::deg2tick(1, qdeg[1]));
  auto t3 = ch::steady_clock::now();

  /* 4 ▸ publish plot data ----------------------------------------- */
  Eigen::Vector2d xy = control::fk_xy(q);   // for visualisation
  Plot p;
  p.stamp       = now();
  p.q1_deg      = qdeg[0];
  p.q2_deg      = qdeg[1];
  p.q1_des_deg  = qd_[0] * 180.0 / M_PI;
  p.q2_des_deg  = qd_[1] * 180.0 / M_PI;
  p.e1          = p.q1_des_deg - qdeg[0];
  p.e2          = p.q2_des_deg - qdeg[1];
  p.x           = xy[0];
  p.y           = xy[1];
  // Add for logger:
  p.measured_q1 = qdeg[0];
  p.measured_q2 = qdeg[1];
```

```
    p.desired_q1  = p.q1_des_deg;
    p.desired_q2  = p.q2_des_deg;
    p.pwm1 = 0.0;
    p.pwm2 = 0.0;
    p.controller_name = "joint_space_idc";
    pub_plot_->publish(p);
    auto t4 = ch::steady_clock::now();

#ifdef PROFILING
    double us_r = ch::duration_cast<ch::microseconds>(t1-t0).count();
    double us_d = ch::duration_cast<ch::microseconds>(t2-t1).count();
    double us_w = ch::duration_cast<ch::microseconds>(t3-t2).count();
    double us_p = ch::duration_cast<ch::microseconds>(t4-t3).count();
    double us_t = ch::duration_cast<ch::microseconds>(t4-t0).count();
    st_[0].update(us_r); st_[1].update(us_d); st_[2].update(us_w);
    st_[3].update(us_p); st_[4].update(us_t);

    if (++loops_ % 100 == 0 || us_t > dt_ * 1e6)
      RCLCPP_INFO(get_logger(),
                  "LOOP#%llu  avgµs [read %.1f | dyn %.1f | write %.1f | pub %.1f |
total %.1f]  worst %.1f",
                  static_cast<unsigned long long>(loops_),
                  st_[0].avg, st_[1].avg, st_[2].avg, st_[3].avg,
                  st_[4].avg, st_[4].worst);
#endif
  }

  /* ── members ──────────────────────────────────────────────────────── */
  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
  rclcpp::Publisher<Plot>::SharedPtr     pub_plot_;
  rclcpp::TimerBase::SharedPtr           timer_;

  dynamixel::PortHandler*   port_{};
  dynamixel::PacketHandler* packet_{};
  dynamixel::GroupBulkRead*  bulk_{};
  dynamixel::GroupSyncWrite* sync_{};

  Eigen::Vector2d qd_{0, 0};   // desired joint angles [rad]
  bool            has_target_{false};

  double kp1_, kd1_, kp2_, kd2_, dt_;
};

/* ── main ─────────────────────────────────────────────────────────────── */
int main(int argc, char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<JointSpaceIDCNode>());
  rclcpp::shutdown();
  return 0;
}
```

Operational Space Inverse Dynamics Controller

```
/* ============================================================================
 * inverse_dynamics_node.cpp — latency-optimised  +  joint-limit clamp
 * ----------------------------------------------------------------------------
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *  Computes     :  planar 2-DOF Operational Space ID  ➜ pwm signal
 *  Writes to    :  Dynamixel PWM Registers
 *  Publishes on :  /idc/joint_plot_data  (controller_msgs/JointPlotData)
 * ============================================================================ */
#include <chrono>
#include <cmath>
#include <algorithm>
#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include <Eigen/Dense>

#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"

#ifndef PROFILING
#define PROFILING 1
#endif

using SetXY = controller_msgs::msg::SetXY;
using Plot  = controller_msgs::msg::JointPlotData;
namespace ch = std::chrono;
using namespace std::chrono_literals;
using std::placeholders::_1;

/* ── Joint hard-limits (deg) ─────────────────────────────────────────── */
constexpr double Q1_MIN_DEG = -90.0;
constexpr double Q1_MAX_DEG =  20.0;
constexpr double Q2_MIN_DEG =  10.0;
constexpr double Q2_MAX_DEG = 140.0;

/* ═══════════════════════════════════════════════════════════════════════ */
class InverseDynamicsNode final : public rclcpp::Node
{
public:
  InverseDynamicsNode() : Node("inverse_dynamics_node")
  {
    /* parameters */
    declare_parameter("port",          std::string{"/dev/ttyUSB0"});
    declare_parameter("baud",          1'000'000);
    declare_parameter("kp1_task",      50.0);
    declare_parameter("kd1_task",       1.0);
    declare_parameter("kp2_task",      50.0);
    declare_parameter("kd2_task",       1.0);
    declare_parameter("ctrl_rate_hz",  500);

    kp1_ = get_parameter("kp1_task").as_double();
    kd1_ = get_parameter("kd1_task").as_double();
    kp2_ = get_parameter("kp2_task").as_double();
```

```
    kd2_ = get_parameter("kd2_task").as_double();
    dt_  = 1.0 / get_parameter("ctrl_rate_hz").as_int();

    /* Dynamixel init */
    port_   = dynamixel::PortHandler::getPortHandler(
                get_parameter("port").as_string().c_str());
    packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
    if (!port_->openPort() ||
        !port_->setBaudRate(get_parameter("baud").as_int()))
      throw std::runtime_error("DXL: cannot open port or set baud");

    /* bulk read: 8 bytes per ID  (vel-pos) */
    bulk_ = new dynamixel::GroupBulkRead(port_, packet_);
    sync_ = new dynamixel::GroupSyncWrite(port_, packet_,
                                          control::ADDR_GOAL_POSITION, 4);
    constexpr uint16_t VEL_ADDR = control::ADDR_PRESENT_VELOCITY; // 128
    if (!bulk_->addParam(control::ID1, VEL_ADDR, 8) ||
        !bulk_->addParam(control::ID2, VEL_ADDR, 8))
      throw std::runtime_error("DXL: GroupBulkRead addParam failed");

    for (uint8_t id : control::SERVO_IDS) {
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
      write8(id, control::ADDR_OPERATING_MODE, control::MODE_POSITION);
      write8(id, control::ADDR_TORQUE_ENABLE, 1);
    }

    /* ROS 2 I/O */
    sub_xy_   = create_subscription<SetXY>(
                  "set_xy", 10, std::bind(&InverseDynamicsNode::cb_target, this, _1));
    pub_plot_ = create_publisher<Plot>("idc/joint_plot_data", 10);
    timer_    = create_wall_timer(
                  ch::duration<double>(dt_), std::bind(&InverseDynamicsNode::cb_loop,
this));

    RCLCPP_INFO(get_logger(), "IDC ready | dt=%.4f s (%.0f Hz)", dt_, 1.0/dt_);
  }

  ~InverseDynamicsNode() override
  {
    for (uint8_t id : control::SERVO_IDS)
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
    delete bulk_;
    delete sync_;
    port_->closePort();
  }

private:
  /* low-level helpers */
  void write8(uint8_t id,uint16_t addr,uint8_t data)
  { uint8_t err=0; packet_->write1ByteTxRx(port_,id,addr,data,&err); }

  bool read_states(double &q1,double &q2,double &dq1,double &dq2)
  {
    if (bulk_->txRxPacket() != COMM_SUCCESS) return false;
    auto ok=[&](uint8_t id,uint16_t a){return bulk_->isAvailable(id,a,4);};
```

```
    if(!(ok(control::ID1, control::ADDR_PRESENT_POSITION ) &&
         ok(control::ID1, control::ADDR_PRESENT_VELOCITY ) &&
         ok(control::ID2, control::ADDR_PRESENT_POSITION ) &&
         ok(control::ID2, control::ADDR_PRESENT_VELOCITY ))) return false;

    int32_t v1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t t1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_POSITION, 4);
    int32_t v2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t t2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_POSITION, 4);

    dq1 = control::vel_tick2rad(v1);
    dq2 = control::vel_tick2rad(v2);
    q1  = control::tick2deg(0, t1) * M_PI/180.0;
    q2  = control::tick2deg(1, t2) * M_PI/180.0;
    return true;
  }

  void send_goal(uint32_t t1,uint32_t t2)
  {
    uint8_t b1[4]={DXL_LOBYTE(DXL_LOWORD(t1)),DXL_HIBYTE(DXL_LOWORD(t1)),
                   DXL_LOBYTE(DXL_HIWORD(t1)),DXL_HIBYTE(DXL_HIWORD(t1))};
    uint8_t b2[4]={DXL_LOBYTE(DXL_LOWORD(t2)),DXL_HIBYTE(DXL_LOWORD(t2)),
                   DXL_LOBYTE(DXL_HIWORD(t2)),DXL_HIBYTE(DXL_HIWORD(t2))};
    sync_->clearParam();
    sync_->addParam(control::ID1,b1);
    sync_->addParam(control::ID2,b2);
    sync_->txPacket();
  }

#ifdef PROFILING
  struct Stat{double avg=0,worst=0;void u(double s){const double
k=.01;avg=(1-k)*avg+k*s;worst=std::max(worst,s);} } st_[5];
  uint64_t loops_=0;
#endif

  /* callbacks */
  void cb_target(const SetXY::SharedPtr m)
  {
    if(!std::isfinite(m->x)||!std::isfinite(m->y)) return;
    target_xy_ = { m->x*0.01, m->y*0.01 };
    has_target_= true;
  }

  void cb_loop()
  {
    auto t0 = ch::steady_clock::now();
    if(!has_target_) return;

    /* 1 ▸ read */
    double q1,q2,dq1,dq2;
    if(!read_states(q1,q2,dq1,dq2)) return;
    auto t1 = ch::steady_clock::now();

    /* 2 ▸ dynamics */
    Eigen::Vector2d q{q1,q2}, dq{dq1,dq2};
```

```
    Eigen::Vector2d x  = control::fk_xy(q);
    Eigen::Matrix2d J  = control::jacobian(q);
    Eigen::Vector2d xd = J*dq;
    Eigen::Vector2d e  = target_xy_ - x;
    Eigen::Vector2d ed = -xd;
    Eigen::Vector2d xdd{ kp1_*e[0]+kd1_*ed[0], kp2_*e[1]+kd2_*ed[1] };
    Eigen::Vector2d qdd = J.inverse() * (xdd - control::jdot_qdot(q,dq));
    Eigen::Vector2d tau = control::mass_matrix(q)*qdd +
                          control::coriolis(q,dq) +
                          control::gravity(q);
    auto t2 = ch::steady_clock::now();

    /* 3 ▸ write (with clamp) */
    Eigen::Vector2d dqdeg = control::tau_to_deg(tau);
    Eigen::Vector2d qdeg  = (q*180.0/M_PI) + dqdeg;

    qdeg[0] = std::clamp(qdeg[0], Q1_MIN_DEG, Q1_MAX_DEG);
    qdeg[1] = std::clamp(qdeg[1], Q2_MIN_DEG, Q2_MAX_DEG);

    send_goal(control::deg2tick(0,qdeg[0]), control::deg2tick(1,qdeg[1]));
    auto t3 = ch::steady_clock::now();

    /* 4 ▸ publish */
    Plot p; p.stamp=now(); p.q1_deg=qdeg[0]; p.q2_deg=qdeg[1];
    p.x = x[0]; p.y = x[1];
    // Add for logger:
    p.measured_q1 = qdeg[0];
    p.measured_q2 = qdeg[1];
    p.desired_q1  = qdeg[0]; // No explicit desired q in operational space, so use
current
    p.desired_q2  = qdeg[1];
    p.pwm1 = 0.0;
    p.pwm2 = 0.0;
    p.controller_name = "inverse_dynamics_node";
    pub_plot_->publish(p);
    auto t4 = ch::steady_clock::now();

#ifdef PROFILING
    double us_r=ch::duration_cast<ch::microseconds>(t1-t0).count();
    double us_d=ch::duration_cast<ch::microseconds>(t2-t1).count();
    double us_w=ch::duration_cast<ch::microseconds>(t3-t2).count();
    double us_p=ch::duration_cast<ch::microseconds>(t4-t3).count();
    double us_t=ch::duration_cast<ch::microseconds>(t4-t0).count();
    st_[0].u(us_r); st_[1].u(us_d); st_[2].u(us_w); st_[3].u(us_p); st_[4].u(us_t);
    if(++loops_%100==0 || us_t>dt_*1e6)
      RCLCPP_INFO(get_logger(),
        "LOOP #%llu  avg_us [read %.1f | dyn %.1f | write %.1f | pub %.1f | total
%.1f]  worst_total %.1f",
        static_cast<unsigned long long>(loops_),
        st_[0].avg, st_[1].avg, st_[2].avg, st_[3].avg, st_[4].avg, st_[4].worst);
#endif
  }

  /* members */
  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
```

```
  rclcpp::Publisher<Plot>::SharedPtr     pub_plot_;
  rclcpp::TimerBase::SharedPtr           timer_;

  dynamixel::PortHandler*   port_{};
  dynamixel::PacketHandler* packet_{};
  dynamixel::GroupBulkRead*  bulk_{};
  dynamixel::GroupSyncWrite* sync_{};

  Eigen::Vector2d target_xy_{0,0};
  bool            has_target_{false};

  double kp1_,kd1_,kp2_,kd2_,dt_;
};

/* ── main ──────────────────────────────────────────────────────────── */
int main(int argc,char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<InverseDynamicsNode>());
  rclcpp::shutdown();
  return 0;
}
```

Robust Joint Controller

```
/* ============================================================================
 * robust_joint_control.cpp  —  Robust Joint-Space Inverse-Dynamics Control
 * ----------------------------------------------------------------------------
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *  Computes     :  planar 2-DOF Operational Space ID  ➜ pwm signal
 *  Writes to    :  Dynamixel PWM Registers
 *  Publishes on :  /robust_joint/joint_plot_data  (controller_msgs/JointPlotData)

 * ============================================================================ */
#include <chrono>
#include <cmath>
#include <algorithm>
#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include <Eigen/Dense>

#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"

using SetXY = controller_msgs::msg::SetXY;
using Plot  = controller_msgs::msg::JointPlotData;
namespace ch = std::chrono;
using namespace std::chrono_literals;
using std::placeholders::_1;

/* ---- joint hard limits (deg) ------------------------------------------- */
constexpr double Q1_MIN_DEG = -90.0, Q1_MAX_DEG =  20.0;
constexpr double Q2_MIN_DEG =  10.0, Q2_MAX_DEG = 140.0;

/* ============================================================================ */
class RobustJointControl : public rclcpp::Node
{
public:
  RobustJointControl() : Node("robust_joint_control")
  {
    /* ---- parameters ------------------------------------------------- */
    declare_parameter("port",          std::string{"/dev/ttyUSB0"});
    declare_parameter("baud",          1'000'000);
    declare_parameter("kp1_joint",     50.0);
    declare_parameter("kd1_joint",      1.0);
    declare_parameter("kp2_joint",     50.0);
    declare_parameter("kd2_joint",      1.0);
    declare_parameter("rho",            5.0);
    declare_parameter("epsilon",        0.01);
    declare_parameter("ctrl_rate_hz",  500);

    kp1_ = get_parameter("kp1_joint").as_double();
    kd1_ = get_parameter("kd1_joint").as_double();
    kp2_ = get_parameter("kp2_joint").as_double();
    kd2_ = get_parameter("kd2_joint").as_double();
    rho_ = get_parameter("rho").as_double();
    eps_ = get_parameter("epsilon").as_double();
```

```
    dt_  = 1.0 / get_parameter("ctrl_rate_hz").as_int();

    /* ---- Dynamixel init (identical to joint_space_idc) --------------- */
    port_   = dynamixel::PortHandler::getPortHandler(
                get_parameter("port").as_string().c_str());
    packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
    if (!port_->openPort() || !port_->setBaudRate(get_parameter("baud").as_int()))
      throw std::runtime_error("DXL: cannot open / set baud");

    bulk_ = new dynamixel::GroupBulkRead(port_, packet_);
    sync_ = new dynamixel::GroupSyncWrite(port_, packet_,
                                          control::ADDR_GOAL_POSITION, 4);
    constexpr uint16_t VEL_ADDR = control::ADDR_PRESENT_VELOCITY;
    if (!bulk_->addParam(control::ID1, VEL_ADDR, 8) ||
        !bulk_->addParam(control::ID2, VEL_ADDR, 8))
      throw std::runtime_error("DXL: GroupBulkRead addParam failed");

    for (uint8_t id : control::SERVO_IDS) {
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
      write8(id, control::ADDR_OPERATING_MODE, control::MODE_POSITION);
      write8(id, control::ADDR_TORQUE_ENABLE, 1);
    }

    /* ---- ROS I/O ----------------------------------------------------- */
    sub_xy_   = create_subscription<SetXY>("set_xy", 10,
                  std::bind(&RobustJointControl::cb_target, this, _1));
    pub_plot_ = create_publisher<Plot>("robust_joint/joint_plot_data", 10);
    timer_    = create_wall_timer(ch::duration<double>(dt_),
                  std::bind(&RobustJointControl::cb_loop, this));

    RCLCPP_INFO(get_logger(),
      "Robust IDC ready  dt=%g s | rho=%g eps=%g", dt_, rho_, eps_);
  }

  ~RobustJointControl() override
  {
    for (uint8_t id : control::SERVO_IDS)
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
    delete bulk_; delete sync_; port_->closePort();
  }

private:
  /* ---- low-level helpers ------------------------------------------- */
  void write8(uint8_t id, uint16_t addr, uint8_t data)
  { uint8_t err{}; packet_->write1ByteTxRx(port_, id, addr, data, &err); }

  bool read_state(double &q1,double &q2,double &dq1,double &dq2)
  {
    if (bulk_->txRxPacket() != COMM_SUCCESS) return false;
    auto ok=[&](uint8_t id,uint16_t a){return bulk_->isAvailable(id,a,4);};
    if(!(ok(control::ID1, control::ADDR_PRESENT_POSITION) &&
         ok(control::ID1, control::ADDR_PRESENT_VELOCITY) &&
         ok(control::ID2, control::ADDR_PRESENT_POSITION) &&
         ok(control::ID2, control::ADDR_PRESENT_VELOCITY))) return false;
```

```
  int32_t v1=bulk_->getData(control::ID1, control::ADDR_PRESENT_VELOCITY,4);
  int32_t p1=bulk_->getData(control::ID1, control::ADDR_PRESENT_POSITION,4);
  int32_t v2=bulk_->getData(control::ID2, control::ADDR_PRESENT_VELOCITY,4);
  int32_t p2=bulk_->getData(control::ID2, control::ADDR_PRESENT_POSITION,4);

  dq1 = control::vel_tick2rad(v1);
  dq2 = control::vel_tick2rad(v2);
  q1  = control::tick2deg(0,p1)*M_PI/180.0;
  q2  = control::tick2deg(1,p2)*M_PI/180.0;
  return true;
}

void send_goal(uint32_t t1,uint32_t t2)
{
  uint8_t b1[4]={DXL_LOBYTE(DXL_LOWORD(t1)),DXL_HIBYTE(DXL_LOWORD(t1)),
                 DXL_LOBYTE(DXL_HIWORD(t1)),DXL_HIBYTE(DXL_HIWORD(t1))};
  uint8_t b2[4]={DXL_LOBYTE(DXL_LOWORD(t2)),DXL_HIBYTE(DXL_LOWORD(t2)),
                 DXL_LOBYTE(DXL_HIWORD(t2)),DXL_HIBYTE(DXL_HIWORD(t2))};
  sync_->clearParam();
  sync_->addParam(control::ID1,b1);
  sync_->addParam(control::ID2,b2);
  sync_->txPacket();
}

/* ---- callbacks ------------------------------------------------------ */
void cb_target(const SetXY::SharedPtr m)
{
  if(!std::isfinite(m->x)||!std::isfinite(m->y)) return;
  double j1,j2;
  if(!control::ik_xy(m->x*0.01, m->y*0.01, j1, j2)){
    RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000,
      "IK fail (%.1f,%.1f)", m->x, m->y);
    return;
  }
  qd_ = {j1,j2};  has_target_ = true;
}

void cb_loop()
{
  if(!has_target_) return;

  double q1,q2,dq1,dq2;
  if(!read_state(q1,q2,dq1,dq2)) return;

  Eigen::Vector2d q{q1,q2},               dq{dq1,dq2};
  Eigen::Vector2d qd = qd_,               dqd = {0,0},  ddqd = {0,0};
  Eigen::Vector2d q_tilde  = qd - q;
  Eigen::Vector2d dq_tilde = dqd - dq;

  /* ---- error state ξ & sliding vector z ---------------------------- */
  Eigen::Vector4d xi; xi << q_tilde, dq_tilde;
  Eigen::Matrix<double,2,4> D_T;  // actually Dᵀ with rows that pick velocities
  D_T.setZero();  D_T(0,2)=1; D_T(1,3)=1;
  Eigen::Vector2d z = D_T * xi;   // Q = I so Dᵀ Q ξ = Dᵀ ξ
```

```
    /* ---- robust term ------------------------------------------------ */
    Eigen::Vector2d w = control::robust_w(z, rho_, eps_);

    /* ---- desired accel (PD+robust) ---------------------------------- */
    Eigen::Vector2d qdd_star;
    qdd_star[0] = ddqd[0] + kd1_*dq_tilde[0] + kp1_*q_tilde[0] + w[0];
    qdd_star[1] = ddqd[1] + kd2_*dq_tilde[1] + kp2_*q_tilde[1] + w[1];

    Eigen::Vector2d tau = control::mass_matrix(q)*qdd_star +
                          control::coriolis(q,dq) +
                          control::gravity(q);

    /* ---- position-mode conversion ----------------------------------- */
    Eigen::Vector2d qdeg = (q*180.0/M_PI) + control::tau_to_deg(tau);
    qdeg[0]=std::clamp(qdeg[0],Q1_MIN_DEG,Q1_MAX_DEG);
    qdeg[1]=std::clamp(qdeg[1],Q2_MIN_DEG,Q2_MAX_DEG);
    send_goal(control::deg2tick(0,qdeg[0]), control::deg2tick(1,qdeg[1]));

    /* ---- publish plot ----------------------------------------------- */
    Plot p;
    p.stamp=now();
    p.q1_deg=qdeg[0]; p.q2_deg=qdeg[1];
    p.q1_des_deg=qd[0]*180.0/M_PI;
    p.q2_des_deg=qd[1]*180.0/M_PI;
    p.e1=p.q1_des_deg-p.q1_deg;
    p.e2=p.q2_des_deg-p.q2_deg;
    auto xy = control::fk_xy(q);
    p.x=xy[0]; p.y=xy[1];
    // Add for logger:
    p.measured_q1 = qdeg[0];
    p.measured_q2 = qdeg[1];
    p.desired_q1  = p.q1_des_deg;
    p.desired_q2  = p.q2_des_deg;
    p.pwm1 = 0.0;
    p.pwm2 = 0.0;
    p.controller_name = "robust_joint_control";
    pub_plot_->publish(p);
  }

  /* ---- members ------------------------------------------------------ */
  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
  rclcpp::Publisher<Plot>::SharedPtr     pub_plot_;
  rclcpp::TimerBase::SharedPtr           timer_;

  dynamixel::PortHandler*   port_{};
  dynamixel::PacketHandler* packet_{};
  dynamixel::GroupBulkRead*  bulk_{};
  dynamixel::GroupSyncWrite* sync_{};

  Eigen::Vector2d qd_{0,0};
  bool   has_target_{false};
  double kp1_,kd1_,kp2_,kd2_,rho_,eps_,dt_;
};

/* main ------------------------------------------------------------------ */
```

```
int main(int argc,char** argv)
{
  rclcpp::init(argc,argv);
  rclcpp::spin(std::make_shared<RobustJointControl>());
  rclcpp::shutdown();
  return 0;
}


       Vision Parameters
#pragma once
#include <opencv2/imgproc.hpp>
#include <vector>
#include <algorithm>
#include <cmath>

/* ── shared vision constants ──────────────────────────────────────── */
namespace vp {
constexpr int    IMG_W = 640, IMG_H = 240;
constexpr int    HALF_W = IMG_W / 2;
constexpr double IMG_CX = HALF_W / 2.0;
constexpr double IMG_CY = IMG_H  / 2.0;

constexpr int    THRESH    = 225;
constexpr int    ERODE_SZ  = 3, DILATE_SZ = 7;
constexpr double MIN_AREA  = 15.0 * 15.0;

constexpr double HFOV_DEG  = 45.0;
constexpr double CAMERA_H  = 85.0;          // camera height [cm]
constexpr double BASELINE  = 5.5;           // stereo baseline [cm]
constexpr double FOCAL_PX  = (HALF_W / 2.0) /
        std::tan((HFOV_DEG * M_PI / 180.0) / 2.0);

/* Physics & filter */
constexpr double G_CM       = 981.0;        // gravity [cm s⁻²]
constexpr double LPF_ALPHA  = 0.7;
constexpr double VEL_ALPHA  = 0.7;
constexpr double PRED_ALPHA = 0.6;
constexpr std::size_t BUF_MAX      = 4;
constexpr std::size_t BUF_MIN      = 2;
constexpr double      RESET_GAP_S  = 0.30;
} // namespace vp

/* ── centroid helper (optional: move to vision_utils.hpp) ─────────── */
inline bool centroid(const cv::Mat& roi,
                     double& cx, double& cy, double& area,
                     const cv::Mat& erodeK, const cv::Mat& dilateK)
{
  cv::Mat g, m;  cv::cvtColor(roi, g, cv::COLOR_BGR2GRAY);
  cv::threshold(g, m, vp::THRESH, 255, cv::THRESH_BINARY);
  cv::erode(m, m, erodeK);  cv::dilate(m, m, dilateK);

  std::vector<std::vector<cv::Point>> cont;
  cv::findContours(m, cont, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
  if (cont.empty()) return false;
```

```
  auto it = std::max_element(cont.begin(), cont.end(),
            [](auto& a, auto& b){ return cv::contourArea(a) < cv::contourArea(b); });
  area = cv::contourArea(*it);
  if (area < vp::MIN_AREA) return false;

  cv::Moments mom = cv::moments(*it);
  cx = mom.m10 / mom.m00;
  cy = mom.m01 / mom.m00;
  return true;
}
```

Control Parameters

```
#pragma once
#include <array>
#include <cstdint>

namespace control
{
/* ================== Hardware ================== */
constexpr std::array<int,2> SERVO_IDS{4,1};   // ID1, ID2
constexpr uint8_t ID1 = 4, ID2 = 1;

/* Dynamixel registers (XM430-W210) */
constexpr uint16_t ADDR_TORQUE_ENABLE   = 64;
constexpr uint16_t ADDR_OPERATING_MODE  = 11;
constexpr uint16_t ADDR_GOAL_POSITION   = 116;
constexpr uint16_t ADDR_GOAL_PWM        = 100;
constexpr uint16_t ADDR_PRESENT_POSITION= 132;
constexpr uint16_t ADDR_PRESENT_VELOCITY= 128;

/* Operating-mode codes */
constexpr uint8_t MODE_POSITION = 3;
constexpr uint8_t MODE_PWM      = 16;

/* ================== Geometry ================== */
constexpr double L1 = 0.30;                 // [m]
constexpr double L2 = 0.30;                 // [m]
constexpr double BASE_X = -0.42;            // [m] world frame
constexpr double BASE_Y =  0.01;            // [m]
constexpr double Q1_MIN = -80.0 * M_PI / 180.0;
constexpr double Q1_MAX = -10 * M_PI / 180.0;
constexpr double Q2_MIN =  20.0 * M_PI / 180.0;
constexpr double Q2_MAX =  140 * M_PI / 180.0;

/* ================== Servo calibration ========= */
constexpr double ZERO_TICK_1 = 2048;        // ticks when q1 = 0 deg
constexpr double ZERO_TICK_2 = 2048;
constexpr double TICKS_PER_DEG = 4096.0/360.0;

/* Velocity conversion (tick/s → rad/s) */
constexpr double VELOCITY_SCALE = 0.229 * 2*M_PI / 60.0;

/* ================== Dynamics ================== */
constexpr double M1 = 0.17;                 // [kg] link 1
constexpr double M2 = 0.088;                // [kg] link 2
constexpr double LC1 = L1*0.7;              // COM offsets
constexpr double LC2 = L2*0.5;
constexpr double I1 = 0.00110;              // [kg·m²]
constexpr double I2 = 0.00030;

/* PWM / torque limits */
constexpr double TAUMAX1 = 2.5;             // [N·m]
constexpr double TAUMAX2 = 2.5;
constexpr int    PWM_LIM = 885;             // full-scale
} // namespace control
```

Control Utilities

```
#pragma once
#include "control_params.hpp"
#include <Eigen/Dense>
#include <cmath>
#include <algorithm>
#include <dynamixel_sdk/dynamixel_sdk.h>

namespace control
{
/* ---------- tick/deg helpers ----------------------------------- */
inline uint32_t deg2tick(int j,double deg){
  const double z = (j==0?ZERO_TICK_1:ZERO_TICK_2);
  return static_cast<uint32_t>(z + deg*TICKS_PER_DEG);
}
inline double tick2deg(int j,int32_t t){
  const double z = (j==0?ZERO_TICK_1:ZERO_TICK_2);
  return (t - z)/TICKS_PER_DEG;
}
inline double vel_tick2rad(int32_t v){ return v*VELOCITY_SCALE; }

/* ---------- FK  (planar) --------------------------------------- */
inline Eigen::Vector2d fk_xy(const Eigen::Vector2d& q){
  double c1=std::cos(q[0]), s1=std::sin(q[0]);
  double c12=std::cos(q[0]+q[1]), s12=std::sin(q[0]+q[1]);
  return { BASE_X + L1*c1 + L2*c12,
           BASE_Y + L1*s1 + L2*s12 };
}

/* ---------- Planar IK (camera XY → joint rad) ------------------- */
inline bool ik_xy(double x,double y,double& q1,double& q2)
{
  const double dx = x - BASE_X;
  const double dy = y - BASE_Y;
  const double r2 = dx*dx + dy*dy;
  const double L12 = L1 + L2;
  if (r2 > L12*L12) return false;                 // unreachable

  double c2 = (r2 - L1*L1 - L2*L2) / (2*L1*L2);
  c2 = std::clamp(c2, -1.0, 1.0);
  double s2 = std::sqrt(1.0 - c2*c2);
  q2 = std::atan2(s2, c2);
  if (q2 < Q2_MIN || q2 > Q2_MAX) return false;

  q1 = std::atan2(dy, dx) - std::atan2(L2*s2, L1 + L2*c2);
  if (q1 < Q1_MIN || q1 > Q1_MAX) return false;
  return true;                                    // always inside limits for now
}

/* ---------- simple PID struct (for PWM node) -------------------- */
struct PidState
{
  double i = 0, prev = 0;
  double step(double err, double kp, double ki, double kd, int max_out)
  {
```

```
    i += ki * err;
    double d = err - prev;  prev = err;
    double u = kp * err + i + kd * d;
    u = std::clamp(u, -double(max_out), double(max_out));
    if (std::abs(u) >= max_out) i = 0;      // crude anti-windup
    return u;
  }
};

/* ---------- blocking read of the two servos (helper) ------------ */
inline void read_two_positions(dynamixel::PortHandler* port,
                               dynamixel::PacketHandler* pkt,
                               int32_t& t1, int32_t& t2)
{
  uint8_t err{};
  pkt->read4ByteTxRx(port, ID1, ADDR_PRESENT_POSITION, (uint32_t*)&t1, &err);
  pkt->read4ByteTxRx(port, ID2, ADDR_PRESENT_POSITION, (uint32_t*)&t2, &err);
}


/* ---------- Jacobian -------------------------------------------- */
inline Eigen::Matrix2d jacobian(const Eigen::Vector2d& q){
  double s1=std::sin(q[0]), c1=std::cos(q[0]);
  double s12=std::sin(q[0]+q[1]), c12=std::cos(q[0]+q[1]);
  Eigen::Matrix2d J;
  J << -L1*s1 - L2*s12 ,  -L2*s12,
        L1*c1 + L2*c12 ,   L2*c12;
  return J;
}

/* J˙q˙------------------------------------------------------------ */
inline Eigen::Vector2d jdot_qdot(const Eigen::Vector2d& q,
                                 const Eigen::Vector2d& dq){
  double dq12=dq[0]+dq[1];
  double c1=std::cos(q[0]), s1=std::sin(q[0]);
  double c12=std::cos(q[0]+q[1]), s12=std::sin(q[0]+q[1]);

  Eigen::Vector2d v;
  v[0]=(-L1*c1*dq[0]-L2*c12*dq12)*dq[0] + (-L2*c12*dq12)*dq[1];
  v[1]=(-L1*s1*dq[0]-L2*s12*dq12)*dq[0] + (-L2*s12*dq12)*dq[1];
  return v;
}

/* ---------- Mass matrix, Coriolis, Gravity ---------------------- */
inline Eigen::Matrix2d mass_matrix(const Eigen::Vector2d& q){
  double c2=std::cos(q[1]);
  double m11 = I1 + I2 +
               M1*LC1*LC1 +
               M2*(L1*L1 + LC2*LC2 + 2*L1*LC2*c2);
  double m12 = I2 + M2*(LC2*LC2 + L1*LC2*c2);
  double m22 = I2 + M2*LC2*LC2;
  Eigen::Matrix2d M; M << m11, m12, m12, m22; return M;
}
inline Eigen::Vector2d coriolis(const Eigen::Vector2d& q,
                                const Eigen::Vector2d& dq){
```

```
  double s2=std::sin(q[1]);
  double h = -M2*L1*LC2*s2;
  return { h*(2*dq[0]*dq[1] + dq[1]*dq[1]),
          -h*dq[0]*dq[0] };
}
inline Eigen::Vector2d gravity(const Eigen::Vector2d& q){
  /* ignored (arm horizontal) → return zeros */
  return {0,0};
}

/* ---------- Torque → small position step (heuristic) ------------ */
inline Eigen::Vector2d tau_to_deg(const Eigen::Vector2d& tau){
  return { tau[0]/TAUMAX1 * (PWM_LIM/20.0),   //  ~±45 deg FS
           tau[1]/TAUMAX2 * (PWM_LIM/2.0) };
}

/* ---------- Robust Control Parameters ------------ */
inline Eigen::Vector2d robust_w(const Eigen::Vector2d& z,
                                double rho, double eps)
{
  const double norm_z = z.norm();
  const double gain   = (norm_z >= eps) ? (rho / norm_z)
                                        : (rho / eps);
  return gain * z;
}

/* ================================================================
 *  Adaptive-control helpers  (planar 2-DOF arm)
 * ================================================================ */
namespace detail
{
/* dynamic regressor  Y(q,q˙,q¨r)  — 2×10, fixed-size, no heap
   π = [α1 α2 α3 α4 α5  α6 α7 α8 α9 α10]^T encodes link masses,
       inertias, COM offsets.  Expressions follow Spong '06.          */
inline Eigen::Matrix<double,2,10> regressor(
        const Eigen::Vector2d& q,
        const Eigen::Vector2d& dq,
        const Eigen::Vector2d& ddqr)
{
  const double c2 = std::cos(q[1]),
               s2 = std::sin(q[1]);

  /* shorthand */
  const double q1dd = ddqr[0],  q2dd = ddqr[1];
  const double q1d  = dq[0],    q2d  = dq[1];

  Eigen::Matrix<double,2,10> Y; Y.setZero();

  /* Row 1 ---------------------------------------------------------- */
  Y(0,0) = q1dd;
  Y(0,1) = c2*q1dd - s2*q2d*q1d;
  Y(0,2) = q2dd;
  Y(0,3) = q1dd;
  Y(0,4) = 0.0;
  Y(0,5) = 2*c2*q1dd + c2*q2dd - 2*s2*q1d*q2d - s2*q2d*q2d;
```

```
  Y(0,6) = q2dd;
  Y(0,7) = 0.0;
  Y(0,8) = 0.0;
  Y(0,9) = 0.0;

  /* Row 2 ------------------------------------------------------------ */
  Y(1,0) = 0.0;
  Y(1,1) = s2*q1d*q1d + c2*q2dd;
  Y(1,2) = 0.0;
  Y(1,3) = 0.0;
  Y(1,4) = q1dd + q2dd;
  Y(1,5) = s2*q1d*q1d + c2*q1dd;
  Y(1,6) = q1dd + q2dd;
  Y(1,7) = 0.0;
  Y(1,8) = 0.0;
  Y(1,9) = 0.0;

  return Y;
}
} // namespace detail

/* ======== small helper that clamps a vector entry-wise ========== */
template<int N>
inline void clamp_vec(Eigen::Matrix<double,N,1>& v, double lim)
{
  for(int i=0;i<N;++i) v[i] = std::clamp(v[i], -lim, lim);
}


} // namespace control
```

MAE C263C Final Project: Catcher Arm

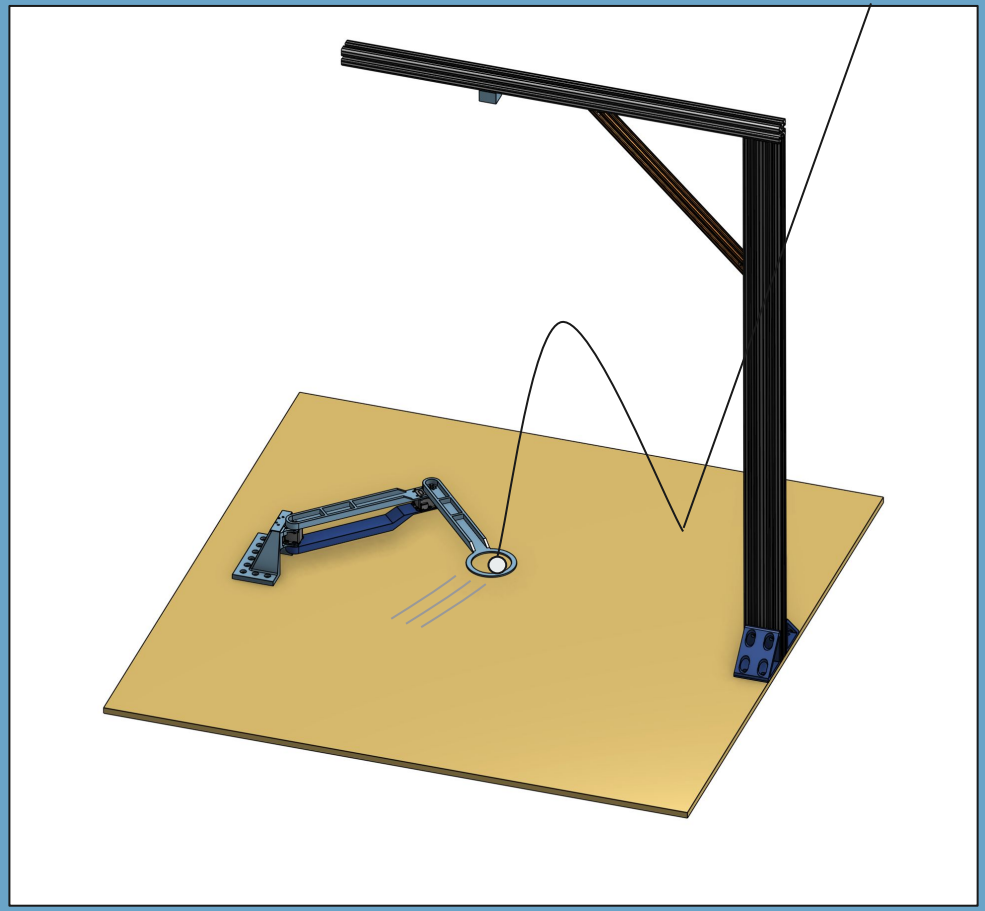
Team 6: Manuel Gonzalez Parra, Justin Lee,
Trevor Oshiro, Eli Whitaker

Overview

Goal: A two-link planar manipulator that can catch a ball in mid air

Motivation: Push these actuators to *need* centralized control through fast, dynamic movements

Applications: Sports, juggling, projectile motion tracking, entertainment

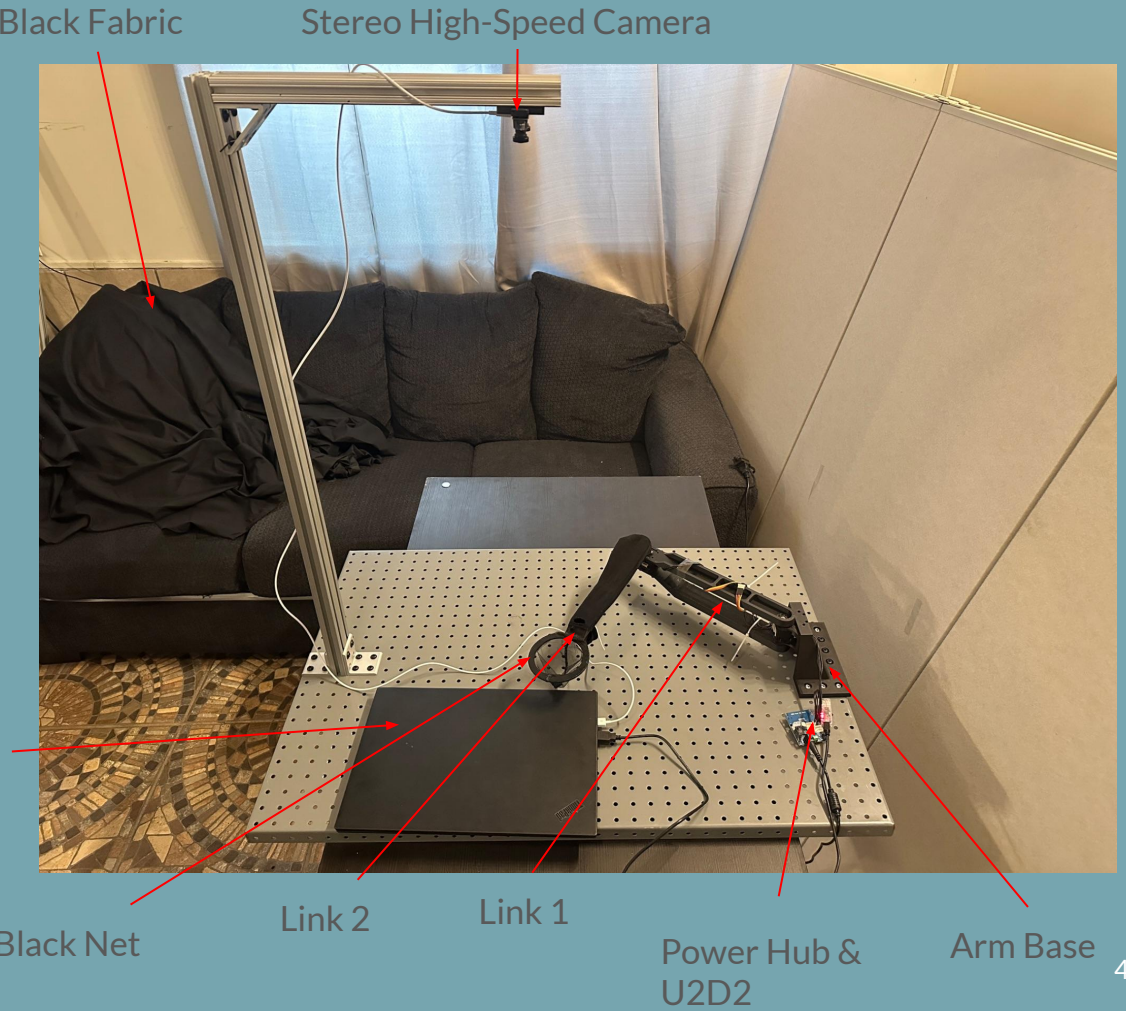


Final Configuration and Results

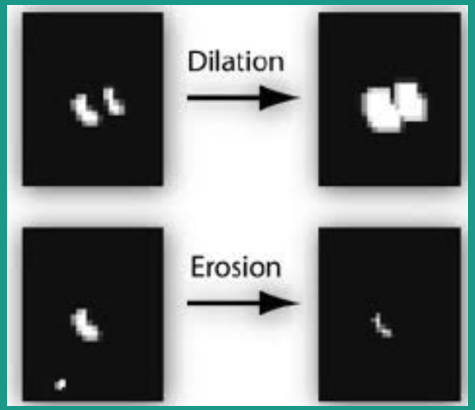
- 2 subsystems
  - Computer Vision used to estimate landing spot
    - Continuously updating target endpoint as opposed to tracking current position
  - Joint Space Robust Controller used to move arm to target
    - Position/velocity based on motor sensors
  - ~75% catch rate

Design Considerations

- Rigid connections between arm and camera for consistent coordinate systems
- Black arm and panel to improve detection of white ball
- Camera mounted high to minimize lens distortion
- Long links needed to be light yet rigid



Methods (Computer Vision)



A human-computer collaborative workflow for the acquisition and analysis by Reda et al.

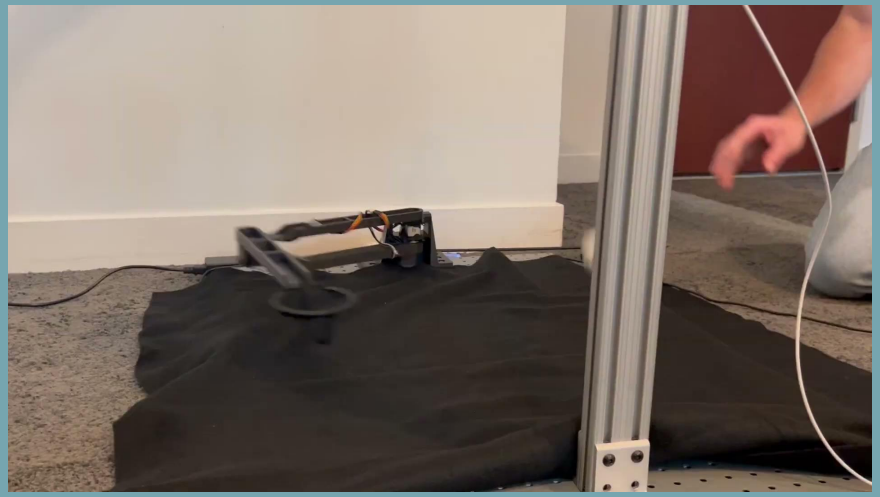
- Centroid of Ball Estimator
  - Binary Threshold
  - Morphological Opening
    - Erosion
    - Dilation
    - Opening
  - Contour Extraction (Suzuki-Abe Algorithm)
    - Finds connected white-pixel boundaries
  - Select Largest Contour
  - Spatial Moments
    - Zeroth Moment
    - First Moments
    - Centroid
- Area Calculation
  - Green's Theorem
- Height estimation
- Projectile Kinematics
  - Filters

Controllers

- First functional system achieved using decentralized joint control
- Implemented and fine-tuned four controller types to increase task success rate
- Each controller contributed unique advantages toward optimizing performance

Controller Type	Pros	Cons
Decentralized Joint Control	• More responsive • Simple to implement and debug	• Jerky movements • Sensitive to noise
Joint space Inverse Dynamics Control	• More responsive relative to robust control • Smaller transient compare do decentralized control	• Computationally intensive • Sensitive to model mismatch or noise
Operational space Inverse Dynamics Control	• Reduces computational load by working in task space • Smaller transient compare do decentralized control	• Difficult gain tuning due to geometry dependence • Highly sensitive to singularities in Jacobian • Prone to cascading errors from inaccurate kinematics
Joint space Robust Control	• Smooth and fluid joint motion • More tolerant to minor disturbances	• Sensitive to jittering • Tends to produce slower settling times

Controller Examples

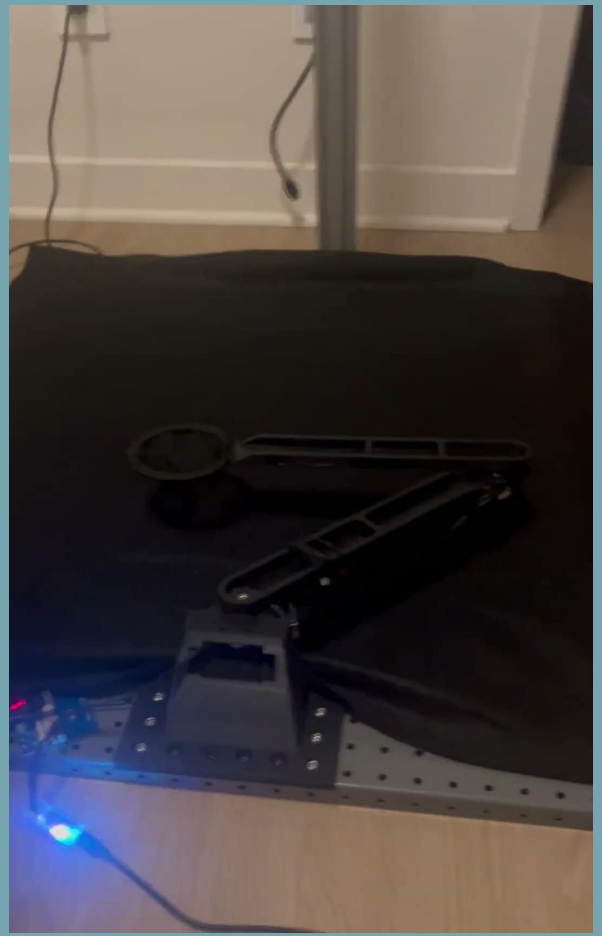


Decentralized PID control

Operational Space Inverse Dynamics

Controller Examples

Joint Space Inverse Dynamics

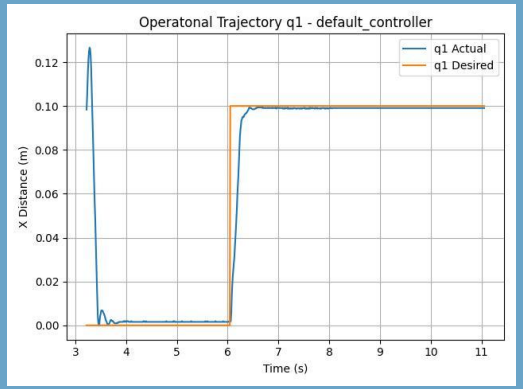


Robust Joint Space Control

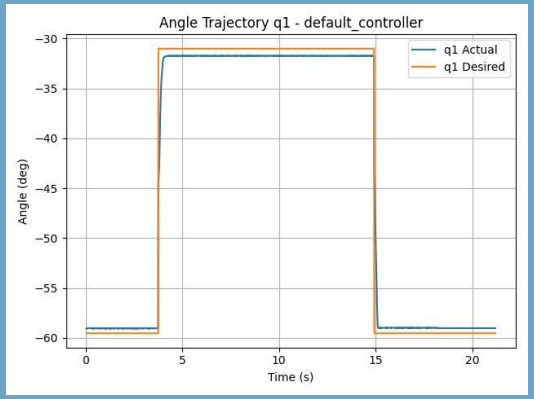
Controller Comparison

- Joint Inverse Dynamics and Decentralized PID best practical performance
- Robust more smooth and oscillation closer to the end point
- Operational Inverse Dynamics sensitive in tuning

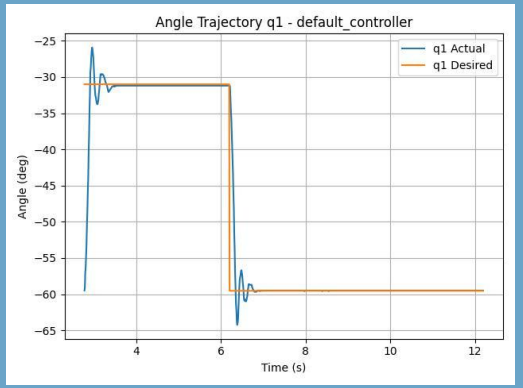
Operational space Inverse Dynamics Control



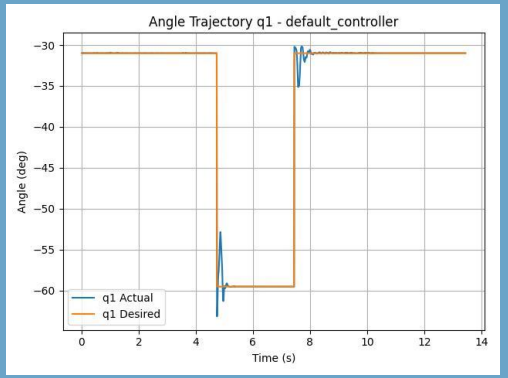
Joint space Inverse Dynamics Control



Decentralized Joint Control



Joint Space Robust Control



Final Results

Successes

- 75% catch rate using final robust controller
- System successfully implemented with full ROS2 integration
- Accurate centroid detection and task generation

Failures

- Lack of sufficient tuning in operational space
- No joint safety limits
- Camera vision conditions and joint position noise reduced precision

Controllers

In order of Pure Qualitative Performance:

1) Robust Control
2) Decentralized Control
3) Joint space Inverse Dynamics Control
4) Operational Space Inverse Dynamics Control

Improvements

- Improve system architecture and streamline gain tuning process
- Add joint limit constraints or emergency stops to prevent damage from overshoot
- Controlled lighting environment (mounted light, reflective taping around ball, infrared camera)