Armlab Report

Shaoze Yang, Rajatsurya M, Jeremy Acheampong
Robotics, The University of Michigan,
Ann Arbor, MI 48109
{shaozey, rajatsm, jacheamp}@umich.edu

***Abstract*—In this multiweek lab for ROB 550 at the University of Michigan, a team of three students programmed a 5-DOF RX200 robotic arm and a RealSense L515 camera to complete five distinct tasks involving object detection, sorting, grasping, stacking colored wooden blocks, and launching balls with a custom-designed mechanism. The lab emphasized integrating camera-captured images with both forward and inverse kinematics to enable the robot to perform these tasks in a competition setting autonomously. Through this project, students gained hands-on experience in computer vision-based object detection, kinematic calculations, and 3D-printed component design, enhancing their robotic manipulation and autonomy skills.**

I. Introduction

In the Armlab, a 5-DOF robotic arm fully autonomously arranges blocks of different colors, sizes, and positions into the desired arrangement. Forward and inverse kinematics are used to determine the appropriate waypoints for our desired end-effector position. A motion planning method was developed to generate feasible waypoints. An overhead Intel RealSense LiDAR Camera is used for block identification on the board. Homogeneous transformations are used to relate pixel and depth coordinates to real-world coordinates. For reliability, the extrinsic matrix is calibrated using four AprilTags with known positions. The hardware used is the Interbotix ReactorX-200 5-DOF Robot Arm, an Intel RealSense LiDAR Camera L515, and an ASUS ROG Laptop with an Intel i7-10750H CPU. Figure 1 depicts the setup of the lab where the RX200 and the L515 camera is secured to a checkered table. Specifically, this 1000mm x 650mm board consists of four April tags and many 50mm x 50mm boxes that are etched into the wood. This is meant to make camera detection easier by adding a grid in the background. For RX200 programming, ROS 2 was used as a platform to transfer information between the camera and the robotic arm platform. OpenCV was also provided as a tool to help with ROS programming. In general, this lab provided a comprehensive introduction to the practical challenges of robotic manipulation, emphasizing the importance of integrating sensing, reasoning, and acting. By the end of the lab, the students had gained valuable experience in building autonomous systems that interact with the physical world, laying the foundation for more advanced robotic applications in the future.

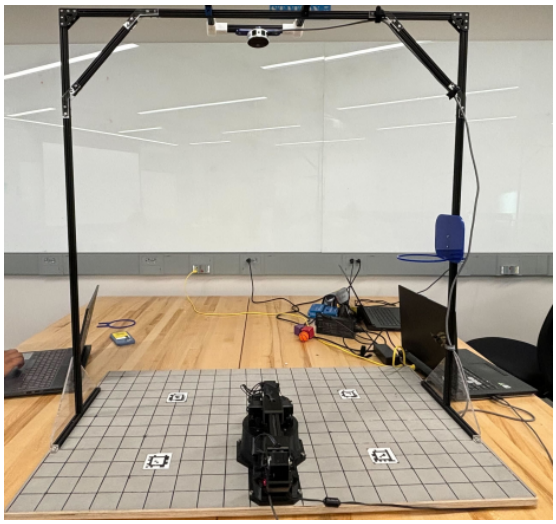


Fig. 1: Image of the setup being used.

II. Methodology

A. Camera Calibration

We used the pinhole camera model to find the location of an object detected in an image in 3D coordinates. In this model, a perspective transformation is used to project 3D points into the picture plane, creating a scene view.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Intrinsic}} \underbrace{\begin{bmatrix} \mathbf{R} \mid \mathbf{t} \end{bmatrix}}_{\text{Extrinsic}} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad (1)$$

The coordinates (Xw, Yw, Zw) represent a 3D point in the world. The coordinates (u, v) denote the projection point in pixels. The principal point, (cx, cy), is located at the image center and the focal lengths, (fx, fy), are expressed in pixel units. The intrinsic matrix does not depend on the scene being viewed. It reflects the internal properties of the camera, such as focal length and principal point, which are fixed. Therefore, once our intrinsic matrix is estimated, it can be used again. The rigid body transformation matrix **[R—t]** are the extrinsic parameters. —It is used to describe the rigid motion of an object in front of a still camera.

1. Intrinsic Calibration: We utilize the ROS camera calibration package to perform easy calibration using a checkerboard calibration target[1].

We demonstrate the average intrinsic matrix after performing the checkerboard calibration 5 times and compare it with

TABLE I: Intrinsic Matrix Comparison

Method	Extrinsic Matrix
Manual Calibration	$\begin{bmatrix} 908.61087 & 0 & 642.59 \\ 0 & 908.59074 & 358.7872 \\ 0 & 0 & 1 \end{bmatrix}$
Factory Calibration	$\begin{bmatrix} 898.36 & 0 & 642.59 \\ 0 & 894.41 & 383.17 \\ 0 & 0 & 1 \end{bmatrix}$

the factory intrinsic matrix as shown in Table I. The focal length parameters in factory calibration are slightly lower than those in manual calibration. The x-coordinates of the principal point are identical, on the other hand, the y-coordinate differs by approximately 24 pixels. These deviations may have been caused by accidental collision, differences in image resolution used for calibration, or even potential lens distortion.

2. Extrinsic calibration - Rough calibration: We first use physical measurements of the camera's position to form a rough extrinsic matrix with the following assumptions:

(a) X axis of the world frame are parallel to U axes of the sensor.

(b) YZ plane of the camera frame is parallel to the YZ plane of the world.

(c) Camera frame is at the front surface of the sensor.

TABLE II: Extrinsic Matrix Comparison

Method	Extrinsic Matrix
Manual Calibration	$\begin{bmatrix} 1 & -3.000 \times 10^{-3} & -2.400 \times 10^{-2} & 1.800 \times 10^{1} \\ 0 & -9.900 \times 10^{-1} & 1.390 \times 10^{-1} & 9.400 \times 10^{1} \\ -2.400 \times 10^{-2} & -1.390 \times 10^{-1} & -9.900 \times 10^{-1} & 1.017 \times 10^{3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
PnP Calibration	$\begin{bmatrix} 9.999 \times 10^{-1} & 8.562 \times 10^{-3} & -2.123 \times 10^{-3} & 1.367 \times 10^{1} \\ 8.732 \times 10^{-3} & -9.949 \times 10^{-1} & 1.003 \times 10^{-1} & 1.536 \times 10^{2} \\ -1.253 \times 10^{-3} & -1.003 \times 10^{-1} & -9.949 \times 10^{-1} & 1.0107 \times 10^{3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$

3. Extrinsic Calibration - Auto Calibration (AprilTags): In this calibration procedure, the camera's extrinsic parameters are determined using AprilTag detections; the process begins with retrieving the AprilTag detection message, extracting each tag's center pixel and corresponding depth, and converting these values into 3D camera coordinates by applying the intrinsic camera matrix, thereby storing both 2D and 3D tag positions in dedicated dictionaries. The algorithm then identifies common tag IDs between the detections and the predefined world coordinates, ensuring a minimum of four correspondences to proceed; if insufficient correspondences exist, the process terminates with an error. Once valid, the 2D image points, 3D camera points, and corresponding world points are compiled into arrays, and a homogeneous transformation matrix is computed using an SVD-based approach to align the camera frame with the world frame, subsequently updating the camera's extrinsic parameters and setting a calibration flag. Additionally, a homography matrix is computed via OpenCV's cv2.findHomography function by mapping detected source points to predefined destination points for color image alignment, and the resulting transformation matrices are printed to confirm successful calibration.

TABLE III: AprilTags Calibration

Method	Extrinsic Matrix
Homogeneous Transformation	$\begin{bmatrix} 9.9996 \times 10^{-1} & 8.5622 \times 10^{-3} & -2.1226 \times 10^{-3} & 1.3673 \times 10^{1} \\ 8.7320 \times 10^{-3} & -9.9492 \times 10^{-1} & 1.0030 \times 10^{-1} & 1.5362 \times 10^{2} \\ -1.2530 \times 10^{-3} & -1.0032 \times 10^{-1} & -9.9495 \times 10^{-1} & 1.0108 \times 10^{3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Homography Matrix	$\begin{bmatrix} 1.1867 & -4.7527 \times 10^{-2} & -1.5926 \times 10^{2} \\ -3.7696 \times 10^{-3} & 1.1505 & -7.6449 \times 10^{1} \\ 1.7437 \times 10^{-5} & -9.4280 \times 10^{-5} & 1.0000 \end{bmatrix}$

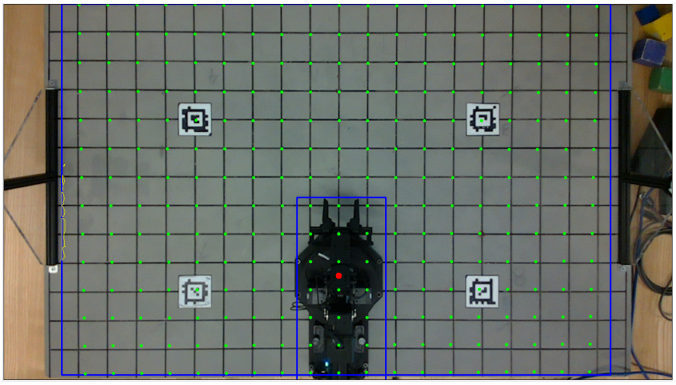


Fig. 2: Image of Projected grid points

The calibration was evaluated by projecting grid points onto the image plane and comparing them to the observed grid intersections. Qualitatively there is a slight shift between the grid points (green dots) and the actual intersections of the grid intersections on the board. The error is about 5 mm. For computing the projective transformation, AprilTags were used defined by each having their ID with a corresponding (x,y) pixel coordinate.

Calibration verification involved the evaluation of the z-axis values by simply moving the mouse across the GUI, showing a limited range (-3 to +2), showing problems with depth estimation. Initially, Perspective-n-Point (PnP) was used for calibration, but it only considered the x and y coordinates, leading to z-axis errors. However, using singular value decomposition (SVD), accounted for all three dimensions and improved the calibration (as seen in Figure 1).

B. Forward Kinematics

Forward kinematics was utilized first to begin controlling the arm. This requires an inputted set of joint angles and will move the end effector to a calculated position. First, a DH table was created where Figure 3 depicts the schematic of the XYZ coordinate frames used to create

TABLE IV: Denavit-Hartenberg parameters of the robot

Link	a_i	α_i	d_i	θ_i
1	0	$-\frac{\pi}{2}$	103.91	$\frac{\pi}{2}$
2	205.73	0	0	-1.32582
3	200	0	0	1.32582
4	0	$-\frac{\pi}{2}$	0	$-\frac{\pi}{2}$
5	0	0	174.15	0

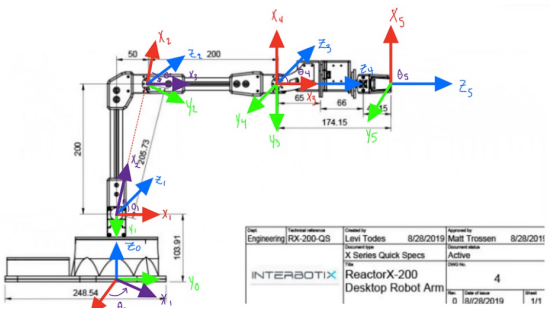


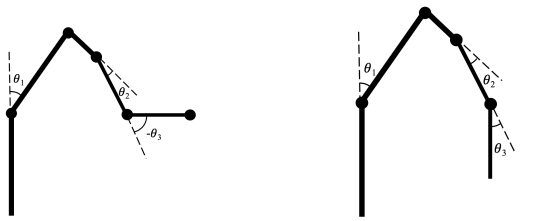
Fig. 3: Image of the setup being used.

Table IV shows the respective DH Table values which are used to create a matrix that calculates the end effector XYZ position. To verify that the forward kinematics worked with our DH Table, we tested it at several known points and checked the calculated end-effector coordinates. Before we made our video for the checkpoint, we kept trying the arm at all four Apriltag positions and the zero position. Essentially, we would turn the torque off and manually put the arm to these coordinates in different configurations to make sure the DH Table worked.

One problem we ran into was that our end-effector location would change based on what configuration the robot was in at a known position. This led to us realizing that the position angle values for 1, 2, and 3 were in the wrong direction. Once the forward kinematics was working at z = 0, we started testing at different heights. We tested at (0, 175, 100), (300, 75, 25), (300, 75, 50), (300, 325, 10), and the zero position. Blocks with known heights were stacked, so the desired z position could be visualized. We could then verify that our end-effector locations were correct by comparing them to the known (x, y, z) points. To check the error, we plugged in the joint angles needed to bring the robot arm to the positions of (0, 175, 100), (300, 75, 25), (300, 75, 50), (300, 325, 10), and (0, 0, 0) and measured how far off the end-effector position was from the base frame. In most cases, the end effector was only off by 5 to 10 mm at most. However, the z position seemed to be proportionally lower the higher and farther the arm reached out. We believe this might be because there is some "sag" in the motors, where when more torque is put on the motor, the more it flexes. However, the error values were not extremely significant (around 15 to 20 mm at most), and we could add an offset angle to increase 3 the higher the z value. As for the typical error, we believe that this could be because the encoders are slightly off in the motors and the robot is built to certain specifications which means it might not be built exactly to the technical sheet data.

C. Inverse Kinematics

We implemented the Inverse Kinematics (IK) to determine the joint positions required to move the end effector to a location in the global frame. Our IK function (in the source file `kinematics.py`, line 464, `IK_analytics()`) that we implemented computes the joint angles from a given transformation matrix that provides the end-effector in Cartesian coordinates. The function also extracts the position coordinates (*x, y, z*) and solves a set of nonlinear equations to determine the most suitable joint angles.



(a) Vertical grasping (b) Horizontal grasping

Fig. 4: Comparison of arm configurations during (a) vertical and (b) horizontal grasping, highlighting the differences in end-effector orientation.

We have two different grasping configurations, vertical grasping and horizontal grasping. These grasping configurations are represented by two sets of nonlinear equations. These equations are derived from the geometric constraints of the arm, as we considered the link lengths and joint positions. The vertical grasping equations obtained from fig 2 are:

$$200\sin(\theta_1) + 50\cos(\theta_1) + 200\sin(\theta_3) - \sqrt{x^2+y^2}$$
$$200\cos(\theta_1) - 50\sin(\theta_1) - 200\cos(\theta_3) - 70.24 - z$$

Similarly, the horizontal grasping equations are:

$$200\sin(\theta_1) + 50\cos(\theta_1) + 200\sin(\theta_3) + 174.15 - \sqrt{x^2+y^2}$$
$$200\cos(\theta_1) - 50\sin(\theta_1) + 200\cos(\theta_3) - 103.91 - z$$

These nonlinear equations are solved using the `fsolve` function from the SciPy library to guess θ_1 and θ_3. The solutions from both configurations are evaluated based on their error norms. The function selects the vertical grasping solution if its error is within 100 times the error of the horizontal solution. The final joint angles are computed by:

Base Joint Rotation:

$$\theta_0 = \tan^{-1}\left(\frac{-x}{y}\right)$$

Elbow and wrist configuration:

$$\theta_2 = \frac{\pi}{2} - \theta_1 - \theta_3 \quad \text{(for vertical grasping)}$$
$$\theta_2 = -\theta_1 - \theta_3 \quad \text{(for horizontal grasping)}$$

Final wrist rotation angle:

$$\theta_4 = \theta_0 \quad \text{(for vertical configuration)}$$
$$\theta_4 = 0 \quad \text{(for horizontal configuration)}$$

In the vertical grasping configuration, the final wrist rotation angle (θ_4), is set to align with the base joint rotation (θ_0). To ensure proper grasping, we fine-tune the gripper's rotation to

be parallel to the block. For the horizontal grasping configuration, no adjustment is needed, so θ_4 remains fixed at 0.

The function ensures precision by verifying that the selected solution has an error norm below 1×10^3. If neither solution meets these conditions, the function flags the result as an error.

D. Motion Planning

Our arm motion plan used the vertical and horizontal grasping configurations shown in Figure 1. The choice of configuration between the two is dependent on the task of the arm. It is important to keep these factors in mind, reachability, joint and torque limits, obstacle avoidance, and dexterity.

The vertical grasping configuration begins with the base rotating to align itself with the target block's location. This initial rotation positions the arm for reaching motion. The arm then extends, through a series of joint movements, to cover the horizontal distance to the block. The final stage of the approach is a vertical descent for the gripper to be positioned 5 cm directly above the block. Once the gripper is correctly positioned, it grasps the block. This configuration is most effective for blocks located close to the robot or directly in front of it, as it minimizes the need for extensive horizontal movement.

The horizontal grasping configuration similarly begins with the base rotating for initial alignment with the target block. However, the primary movement for reaching the block is a horizontal extension. The gripper maintains a horizontal orientation, allowing it to approach the block from the side rather than the top. Once the arm has extended sufficiently to place the gripper at the correct horizontal distance, the gripper closes to grasp the block. This horizontal grasping configuration is better suited for blocks that are further away, requiring the arm to extend significantly.

E. Block Detection

The block detector we implemented is capable of robustly detecting blocks' location (both 2D and 3D coordinates), color (red, orange, yellow, green, blue, purple), size (large and small), and orientation (angle with y-axis) in real-time with color classification accuracy. The result is shown in figure 5.The steps followed to detect blocks are as follows:

1) Preprocessing and Thresholding: The depth data is masked to filter out areas not of interest. Thresholding is applied to isolate the blocks based on depth values. Specifically, the code uses depth thresholds to create a mask (create masked threshold) and applies morphological operations to clean up the image.

2) Contour Detection: After applying the threshold, contours are detected in the masked image. These contours represent potential block candidates

3) Shape and Color Filtering: To refine the detected contours, the code checks if the contours form squares using the aspect ratio (perfect square contours) and the contour's curviness (identify square contours). Blocks are also filtered based on their color. A circular mask is used to retrieve the mean color at the center of the contour, and this is compared with predefined HSV color ranges for various block colors (retrieve area color). This reduces false positives by ensuring that only contours with the correct color are considered blocks.

4) Depth and Area Analysis: The area of each contour is calculated to differentiate between large and small blocks. Depth values at the center of the contours are analyzed to check if the block is stacked or not.

5) Block Characterization: For each detected block, its properties (such as size, color, orientation, and whether it is stacked) are stored in a dictionary. The center, bounding box, and angle of each block are computed, and the block's details are drawn on the image.

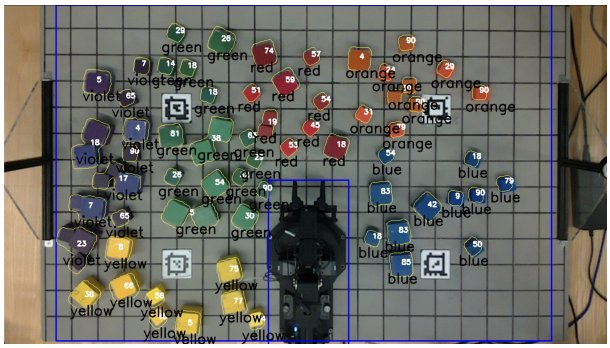


Fig. 5: Detection with Color Classification

Strategies to Limit False Positives and Enhance Robustness:

1) Depth Thresholding: Limiting the depth range ensures that only objects within a certain distance are detected, reducing noise from the background.

2) Shape Filtering: By analyzing the shape and ensuring that only square-like contours are considered, the system filters out irregular objects.

3) Color Matching: Comparing the detected color with predefined color ranges ensures that the detected objects match the expected block colors.

4) Size Filtering: The code filters out blocks that are too small to be real blocks based on contour area.

5) Outlier Rejection: The system rejects contours that deviate significantly in terms of depth or curviness, which helps reduce noise and misdirections.

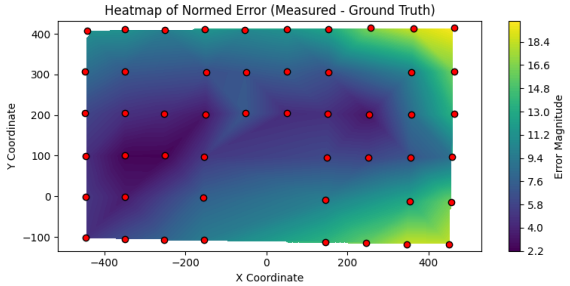


Fig. 6: Heat map of Normed Error

To verify the accuracy of our block detector, we conducted an analysis comparing its output to a manually annotated

ground-truth dataset to visualize the uncertainty in block location across the board. Fig. 6 illustrates the general position errors. The color gradient on the heat map represents the magnitude of the error-warmer colors mean larger errors. Figure 6 illustrates the height errors (Z-axis), which is important for tasks that need vertical positioning.

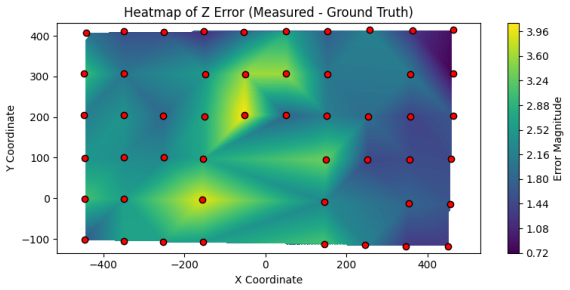


Fig. 7: Heat map of Z-axis Error

These visualizations give us an overview of our block detection system's performance, highlighting areas of high uncertainty that need improvement.

F. Error Compensation

Error Compensation in Block Picking: In our robot arm's block-picking algorithm, we compensate for errors in position estimation to improve accuracy. Given a target position (x, y, z), we use forward kinematics to compute the end-effector position. We account for a constant depth error from the Real Sense camera by scaling the measured z:

$$z' = \max(z, 0) \times \frac{10}{9}$$

Due to the gravity of the robot arm being lower than the target position, we included gravity compensation which varies linearly with the distance from the base of the robot arm, our origin. It accounts for arm deflection, modeled as:

$$\begin{aligned}\text{Error (mm)} = &-0.000001 \cdot (\text{Radius Distance (mm)})^3 \\ &+ 0.000574 \cdot (\text{Radius Distance (mm)})^2 \\ &- 0.07 \cdot \text{Radius Distance (mm)} + 0.06\end{aligned}$$

Therefore, we can represent the final compensated z-position as:

$$Z_{\text{final}} = z' - \Delta z$$

We applied a minimum z-height constraint of 10 mm to avoid a collision between the robot arm and the board.

Error Compensation in Vertical Grasping Configuration: In this vertical grasping configuration, error compensation is important for accurate block placement. As shown in Figure 4, the primary source of error is the robot arm's deflection due to gravity, which varies non linearly with the distance from the base. The graph below illustrates the relationship between our gravity compensation error and radial distance:

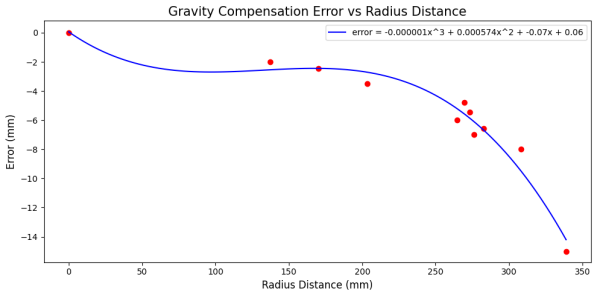


Fig. 8: Overall Gravity Compensation Error vs. Radius Distance

This highlights the need for adjustments in the vertical grasping configuration to counteract the effects of gravity on the arm's performance.

G. Basketball Launcher

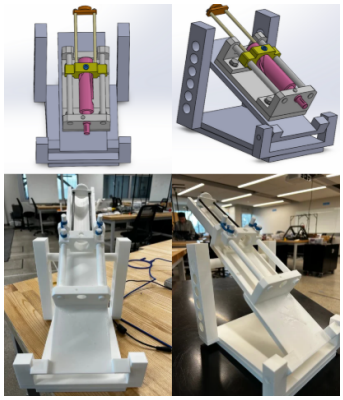


Fig. 9: Basketball launcher mechanism

The basketball launcher mechanism is an elastic band-powered system designed to launch a basketball into a basketball hoop. As seen in Figure 9, it consists of a sturdy 3D-printed base frame to provide stability for the launcher. The adjustable side supports allow for the launch angle to be adjusted, optimizing the trajectory control. The launch mechanism includes an elastic band-loaded actuator (pink component), which generates the force needed to propel the basketball. This actuator is mounted within guiding rails to help maintain alignments and ensure a consistent launch path. A yellow bracket assembly is used to hold the basketball in place before launch. It is connected to elastic bands to store and release energy when triggered. These components, such as the actuator, bracket, hold, and rails, are also 3D printed. Using Figure 10, the initial velocity (Equation (1)) can be found,

$$v_0 = \sqrt{2g(h_1 - h_2) + \frac{k}{m}\Delta s^2} \tag{1}$$

with v_0 as the initial velocity, g as the gravitational constant, k as the spring constant of the rubber bands, m as the mass of the ball, Δs as the change in the length of the rubber bands, and h_1 and h_2 as the two heights of the ball. Therefore,

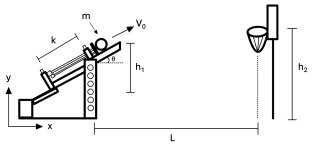


Fig. 10: Illustration depicting the dynamics and mechanics of the basketball launcher in action.

the initial velocity can be calculated based on how much the catapult is pushed downwards.

Using Figure 9(c), the height that the basketball hoop needs to be at can be calculated using Equation (2),

$$h_3 = L\tan(\Theta) - \frac{gL^2}{2(v_0\cos(\Theta))^2} + h_2 \tag{2}$$

where h_3 is the height at which the hoop is positioned, L is the distance from the catapult to the hoop, and Θ is the angle at which the ball is fired. With these trajectory equations, the hoops can be set up so that the exact amount of potential energy needed for every shot is known.

Another part of this design involves a base that can easily be rotated by the arm. Therefore, not only can the arm control the initial velocity of the shot by adjusting how far it pushes down, but it can also control the yaw angle of the shots. Level 3 in event 4 can theoretically be completed.

III. Results and Discussion

A. Teach and Repeat

*1. Strategy:*Teach and repeat is a technique to program robotic arms to perform repetitive tasks. We implemented a "teach-and-repeat" feature to teach the robot to cycle swapping blocks at locations (-100, 225) and (100, 225) through an intermediate location at (250, 75). These coordinates are in units of millimeters (mm), relative to the world frame. We manually placed the robot arm on the positions we need, and record the joint angles that are required to reach that position. Figure – demonstrates this process. We manually added a 'high' position for both the pick and place processes before picking or dropping the blocks, to stabilize the position of the blocks, as well as an intermediate point between each required position (for example, an intermediate point between (100, 225) and (100, 225)). Our robot could swap the blocks 8 times before it failed. Fig. 12 shows the joints position, velocities and efforts over time during this process.

Problems and Improvements The 9th attempt failed because the block placement was off during the process, preventing the robot arm from successfully picking up the blocks. Rather than manually recording the joint angles for positioning, we can utilize inverse kinematics(IK) to automatically generate the required joint angles.Additionally, implementing a block detection technique can help address misalignment issues, ensuring more accurate block picking.

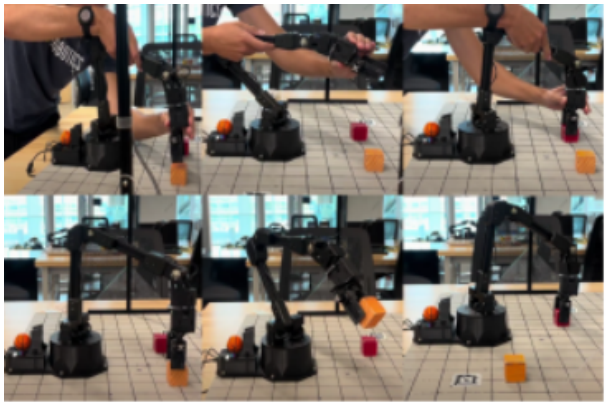


Fig. 11: Teach and repeat process demonstration where the top pictures are the "taught" waypoints and the bottom pictures are the repetitions.

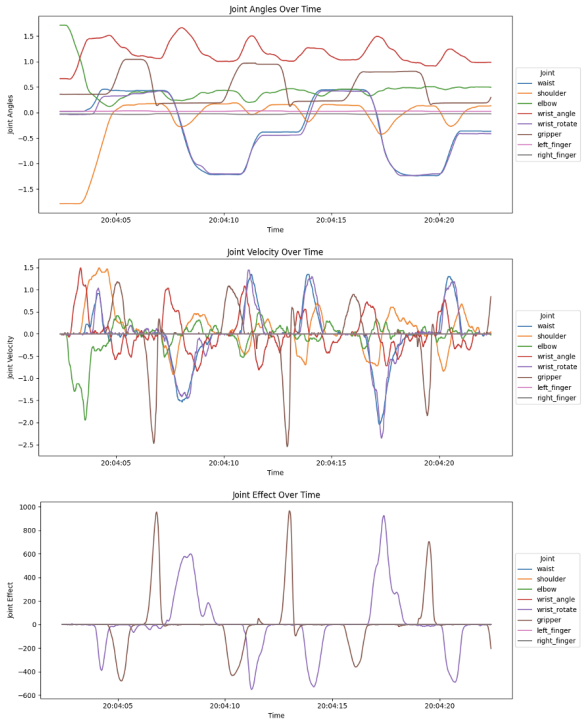


Fig. 12: Robot joint positions, velocities, and efforts over time during teach and repeat block swapping.

B. Events of the Competition

1) Sort 'n Stack

Strategy Our approach leverages the pre-built BlockDetector to handle most tasks. Initially, the robot arm performs a swapping action at a height of 70 to unstack the large blocks and at 45 for the small blocks. Once unstacking is complete, the robot moves to a sleep position to prevent interference, and the BlockDetector is activated to analyze each block's position, color, and type. The system then organizes and stacks the larger blocks in a rainbow color order, followed by the

smaller blocks.

Problems and Improvements Challenges arise from angle inaccuracies during unstacking, as the combined height of two small blocks nearly matches that of one large block. Additionally, small blocks are often not captured in the correct configuration, occasionally causing stacks to topple. To improve performance, the unstacking process could be refined by having the arm extend directly and follow a set of waypoints to effectively swipe all blocks off their stacks. Moreover, developing new angled grippers that accommodate RX200 variations and guide blocks toward the center—even with slight misalignments—would enhance block trapping.

2) Line 'em Up!

Strategy: We recorded precise waypoints to align all six large and six small blocks in a straight line. The unstacking logic was then applied—executed twice to accommodate potential three-block stacks. Once unstacked, the recorded waypoints guided the sequential pickup and repositioning of each block in line.

Problems and Improvements Relying on predetermined waypoints makes error recovery difficult. Friction on the board sometimes causes blocks to stick during movement, disrupting their intended order. Additionally, if a block is missed at a designated waypoint, the entire sequence is affected. To address these issues, future improvements include elevating the waypoints to prevent board contact and employing angled grippers designed to correct block grabbing errors.

3) To the Sky!

Strategy For this event, we hard coded the positions of the robot arm for both the pickup and placement of blocks, employing a three-step stacking strategy. The block pickup position was fixed at (0, 200, 30), while the placement position was set at (-255, 0, h), where h represents the current height of stack 1.

1) Place five blocks with an orientation angle $\theta = \pi/2$.
2) Begin a new stacking sequence at (300, 100, h_1), where h_1 is the height of stack 2. The robot arm then lifts stack 2 from the bottom and places it on top stack 1.

To prevent stack 2 from collapsing, the robot arm's movement speed was set to 5. While a maximum height of 16 blocks was achieved in testing, it proved unstable.

Problems and Improvements

The inverse kinematics (IK) solver can generate joint angles that are mathematically feasible but physically unattainable due to the mechanical constraints of the robot arm. When stacking directly onto stack 1, the constraint of robot arm limits the stacking height to approximately 13–14 blocks. Stacking more than 14 blocks requires a negative θ, which leads to instability.

Further challenges arise when lifting and positioning stack 2 on top of stack 1. Stack 2 is prone to collapse during lifting and improper placement, either too high or too low, can cause stack 1 to be crushed.

To mitigate these issues, we propose the following improvements:

1) Enhancing the IK Solver: Improve the inverse kinematics function to handle placements that require angles near $\theta = \pi/2$, where failures are the most common.
2) Optimization of intermediate parameters: Fine-tune block pickup positions, intermediate poses, arm acceleration profiles, and movement times to increase stability when lifting and stacking stack 2.
3) Refine the placement strategy: Adjust the final placement position of stack 2 on stack 1 to ensure proper alignment and prevent instability or damage to the stacked structure.

4) Free Throw

Strategy:In this event, the robot had 180 seconds to load and shoot basketballs into hoops using the basketball launcher. The process began with detecting the ball as a block and picking it up with the arm. Way-points were then set using the teach-and-repeat method, guiding the arm to load the ball into the mechanism. The goal was to minimize arm usage and make the mechanism as autonomous as possible to maximize the number of baskets scored within the limited time. We designed a cam mechanism driven by a motor, but the cam profile was too steep, and the motor lacked sufficient torque, resulting in only 15% of the shots being successful.

Problems and Improvements: Our cam profile was too aggressive, and we did not have a motor strong enough to rotate it, causing the cam follower to jam. As a result, we struggled to score baskets. However, when we manually launched the ball using the mechanism, we successfully scored 108 baskets in 180 seconds, achieving maximum points in the competition despite a 70% reduction. To improve performance, we can focus on refining the cam profile for smoother operation and upgrading the motor to provide higher torque.

IV. Conclusion

Throughout this lab, three students explored object detection using camera vision, computed forward and inverse kinematics, and designed 3D-printed components to interface with a robot. By integrating camera calibration, block detection, and inverse kinematics, the robotic arm autonomously grasped blocks on the board and placed them in designated locations. Additionally, incorporating a custom-designed basketball launcher using CAD and forward kinematics enabled the arm to successfully shoot basketballs into hoops. While the programming strategies performed well, several improvements could enhance future performance, such as refining block detection to reduce distortion when blocks are far from the board's center and implementing a more forceful method for unstacking blocks, like extending the arm and sweeping them away using predefined way-points. Hardware refinements, including redesigned angled grippers for better grasping and motor recalibration for improved accuracy, could further optimize performance. Lastly, redesigning the basketball launcher for smoother cam and pin movement, allowing autonomous compression and release of springs or elastic bands, and incorporating a higher torque motor could significantly improve the efficiency of the mechanism.