Autonomous Object Manipulation and Mini-Basketball Launching with a 5-DOF Robot Arm

Justin M. Lu, Chenhuan Jiang, Mingtian Tan
{lujust, chenhj, teemty}@umich.edu

***Abstract*—This paper details the use of a 5-degree-of-freedom Trossen Robotics RX-200 robot arm for precise object manipulation. The system integrates an Intel RealSense LiDAR Camera L515, forward and inverse kinematics, and Apriltag-based camera calibration to automatically align frames. We demonstrate the arm's autonomy by stacking and sorting colored blocks, as well as controlling a custom mini-basketball launcher to accurately shoot balls into a small hoop.**

Results show that combining robust computer vision with closed-form kinematics yields reliable performance for both block manipulation and basketball shooting. This work outlines the methodologies used, the challenges encountered, and the insights gained from a real-world implementation of these robotics concepts.

I. Introduction

Robot arms are crucial in modern automation and robotics, playing a vital role in industries ranging from manufacturing to healthcare. These robotic manipulators enable precise and repeatable tasks that would be difficult or unsafe for humans to perform. In the ROB550 arm lab, we explored the autonomy of a 5-degree-of-freedom robotic arm RX-200 with the Intel RealSense LiDAR Camera L515, focusing on integrating sensing, acting, and reasoning to enable the robot to achieve various complex tasks. This report outlines the methodologies applied in executing these tasks, along with an analysis of the experimental results, highlighting the processes involved in achieving autonomous manipulation and control.

The working environment is shown in Fig.1. The environment consists of a working board, the RX-200 robotic arm and the Realsense Camera above. The robotic arm operates on the board marked with a 50 mm grid, allowing for precise spatial positioning and easy frame definition. The robot is positioned at a defined origin on the grid, with the +z axis extending vertically out from the board. Apriltags are located at four preset positions, which are used for camera calibration with the attached camera. The camera provides both color and depth information and the arm could grab, move and place object with well-defined forward-inverse kinematics, enabling the robotic arm to autonomously perform tasks such as stacking blocks, sorting objects, and other similar operations with precision and efficiency

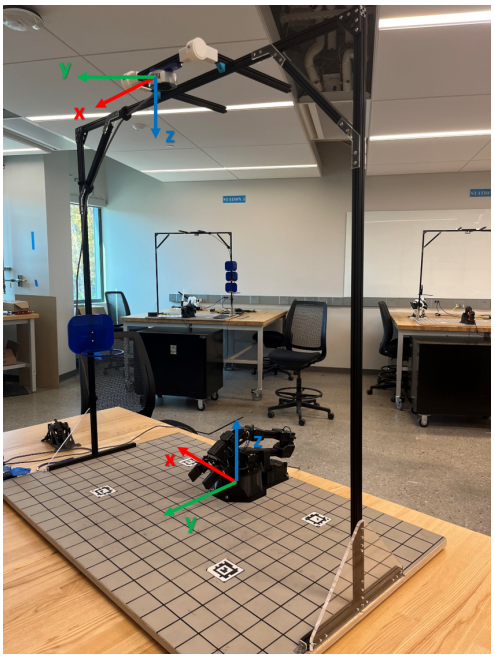


Fig. 1. The workstation of the ROB 550 Armlab

Forward kinematics (FK) is utilized in order to determine the position and orientation of the end-effector based on the joint parameters of the RX-200 arm. Forward kinematics is essential to robotics as it is a method of concretely determining the position and orientation of an end-effector in its operating space with respect to the configuration of a robot's joints.

Inverse kinematics (IK) is a critical concept in robotics, focusing on determining joint parameters required to position a robot's end-effector at a specific location and orientation. Unlike forward kinematics, which calculates the pose from joint variables, IK solves

the reverse problem: finding the joint angles that achieve a desired target position.

For simpler systems, such as a 2-link planar arm (RR manipulator), IK can often be solved using closed-form solutions that rely on geometry (and often, trigonometry). However, for more complex robots with higher degrees of freedom (DOF), analytical solutions are not always feasible. Our approach tackles the 5-DOF IK problem by dividing it into smaller, manageable parts.

We begin by directly calculating the first joint angle, based on the desired position of the end-effector in the horizontal plane. For joints 2, 3, and 4, we treat the problem as a 2-link RR arm and solve the position IK using numerical methods. These joints handle the spatial positioning of the end-effector, while the orientation is decoupled. The fifth joint, which controls the end-effector's orientation or approach angle, is provided as user input.

To solve the IK numerically, we use methods like Newton-Raphson, which iteratively refines joint angles to minimize the difference between the desired and current end-effector positions. This is achieved by employing the Jacobian matrix, which provides a first-order approximation of the system's behavior. When the Jacobian is not invertible, we use the Moore-Penrose pseudoinverse to approximate a solution.

II. Methodology

A. Visual Detection

The visual input is provided by the Intel Realsense LiDAR Camera L515, which transfers RGB-D data to the computer to accurately determine the position, color, and shape of objects on the working board. This enables the robotic arm to utilize this information for automated tasks.

Before object detection, the camera system must be calibrated to update its intrinsic and extrinsic matrices. Once calibrated, object positions and shapes are extracted from the depth data by detecting their contours. The color information is then determined by analyzing the area near the center of each contour.

1) Camera Calibration: Under the assumption that the camera follows a simple 'pinhole' model, two steps are required to transform 3D world coordinates into 2D image coordinates. The first step is to convert the 3D world coordinates into the camera's 3D coordinate system via the camera extrinsic matrix. The second step applies the camera intrinsic matrix to project the 3D camera coordinates onto the 2D image plane.

To perform the initial transformation, the camera's location and orientation (pose) relative to the world coordinate system must be known. This information is represented as a 3D homogeneous transformation -

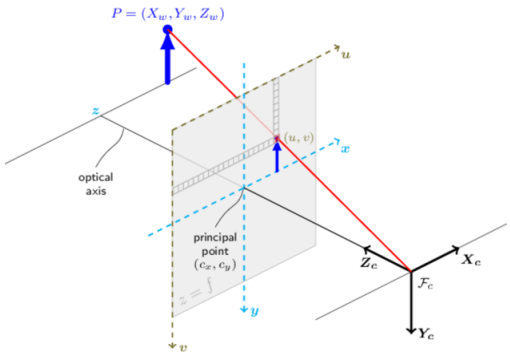


Fig. 2. The pinhole model of the camera system

specifically the aforementioned extrinsic matrix, which consists of a rotation matrix and a translation vector that together describe the camera's pose in world coordinates. Construction of the camera extrinsic matrix is shown in Eq.1:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R_c & p \\ \mathbf{0} & 1 \end{bmatrix} \times \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{1}$$

There are two primary methods to obtain the extrinsic matrix: the first involves manually computing the transformation and fine-tuning parameters to improve accuracy; the second method uses Apriltags and built-in OpenCV functions to automatically calculate the extrinsic matrix.

As shown in Fig.1, the camera coordinates is the world coordinate rotate along the x axis about 180 degrees, and the translation could be measured by measuring tape. The extrinsic matrix could be written as

$$R_c = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta \end{bmatrix}, p = [x, y, z]^T \tag{2}$$

But the camera coordinates is tilted in certain degree which is hard to measure, the only way to eliminate the error is to manually change the θ value until the computed camera z coordinates is relatively consistent on the whole workspace. Then adjust the corresponding x, y, z value to have accurate transformation.

Even though we could get a precise extrinsic matrix by manually adjust it, it would be more efficient to compute it automatically. Utilizing the **cv2.solvePnP()** function which only needs the image coordinates, the corresponding z coordinates and world coordinates and the intrinsic matrix. Since the positional information could be easily obtained for the Apriltags' center, we could compute the extrinsic matrix automatically as long as the four Apriltags are within the image and the intrinsic matrix is known.

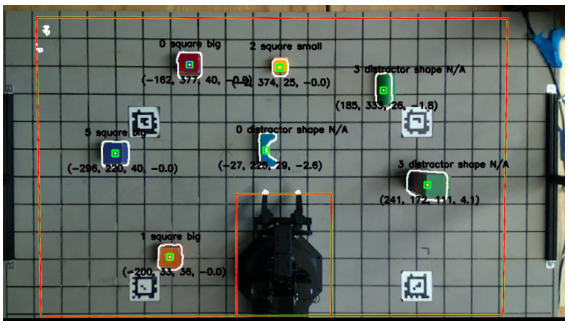


Fig. 3. Example of block detection without layer-by-layer detecting

In Fig.2, the pin hold model shows the detail of how the intrinsic matrix is coming from. It contains the information of the camera, including the focal lengths f_x, f_y, the principle point offset x_0, y_0 and the axis skew s. With these parameters, the transformation from camera coordinate to image coordinates could be expressed as:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z_c} \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & | & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \quad (3)$$

The manual way to obtain the intrinsic matrix is complicated, we need a checkerboard and utilize the **camera_calibration** package, once calibration process starts, we need to manually move the checkerboard according to the status message on the calibration GUI. After that, the built-in function would compute the intrinsic matrix, however, this method is troublesome and the result would vary if the checkerboard moves differently. Instead of using the manually computed intrinsic matrix, we prefer to use the intrinsic matrix provided by the factory, which is simpler and more reliable.

2) Homography transformation: The camera mounted above the workspace is slightly tilted, allowing it to capture the entire area but resulting in a non-rectangular image in the video frame. To streamline the process of locating objects, we can apply a projective transformation to adjust the perspective, converting the trapezoidal view into a rectangle. The transformation could be expressed as left multiply a 3x3 matrix with the image coordinates as in Equation(4), and the 3 by 3 matrix, homography matrix, could be obtained by the provided function **cv2.findHomography()**, which requires 4 image points in original coordinates and their corresponding image coordinates at where you want to place them after transformation.

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (4)$$

Additionally, the homography transformation can be employed to correct the minor misalignment between the RGB and depth images, as the camera and LiDAR are positioned slightly different. after several testing, we found that this misalignment is negligible for the robotic arm to accurately perform tasks such as picking and placing.

3) Block detection: In order to detect the blocks placed inside the workspace, it is required to know the color, shape and positional information from RGB and depth images to properly perform tasks such as sorting. For this project, we decided to utilize the depth image to detect the shape, size and position of the objects and RGB image to determine the color.

Before utilizing the depth image, the entire image needs to be leveled due to the camera not being perfectly aligned vertically. The leveling process is carried out using the provided **transform_images()** function. This process involves first converting the depth coordinates to 3D coordinates in the camera frame, then applying a transformation to reorient the 3D points to match the desired camera pose in the world frame. Finally, the transformed points are mapped back to the depth coordinates. After this process, the depth image becomes uniformly flat, ensuring that the z-axis values remain consistent across the entire image.

After leveling the depth image, we defined a mask to filter out unwanted areas such as the area outside the working board and the area of the arm when it is at the sleep mode. Then the **cv2.bitwise_and()** and **cv2.inRange()** functions could filter out the area where the z value is within the threshold we set. After converting the depth image into the binary image, we apply the **cv2.findContours()** to detect the contours of the objects. Moreover, the we would neglect the contours which has small area size to erase misreadings in some area. Because the objects we wanted is only square blocks, we need to filter out the objects in other shapes like rectangle, semicircle, etc. For shape detection, we approximate the contour with a polygon using **cv2.approxPolyDP()** and assess the number of vertices. If the polygon has 4 or 5 vertices (for small square, it is likely to have 5 vertices due to the accuray of the camera) and its height-to-width ratio is close to 1, we classify the object as a square block, otherwise, it is considered a distractor. For confirmed square blocks, we calculate the contour area to differentiate between small and large blocks for future tasks.

The positional information includes the (x, y, z) position of the center and the orientation (rotation) of the blocks. The center position could be easily obtained by momentum method provided by **cv2.moments()** and the orientation could be obtained by **cv2.minAreaRect()**.

Before collecting the color information, we will first convert the RGB image into HSV image to make the

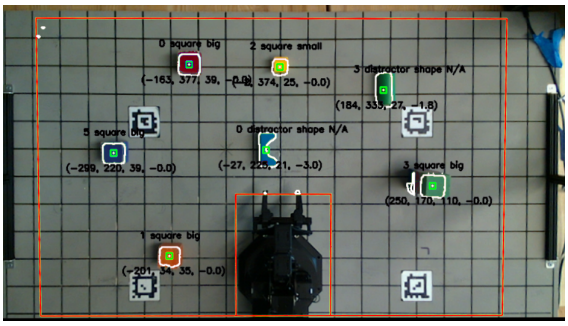


Fig. 4. Example of block detection using layer-by-layer detecting

detection more robust. To detect the object's color, we extract a small region around the center, calculate the mean value, and determine the color by checking which predefined range the hue value falls into. Fig.3 shows an example of detection, and since we need to sort the blocks by color, it's useful to assign a numerical value to each color based on the desired order. In this project, we sort the colors in rainbow order, with 0 corresponding to red, 1 to yellow, and so on, progressing through the spectrum to purple.

However, the above detection in Fig.3 is not perfect when the blocks are stacked (the right green one), the detected contours would deform since it will detect the side of the blocks. Accordingly, we designed a layer-by-layer contour detection function which detects the contours with gradually lower thresholds. The detailed process is first set a large high threshold so the function would detect the highest objects, then mask out the detected area to prevent it from detecting it again in lower threshold. After that, repeat this process for each set value to complete one detection. Fig.4 shows the detected contours and compared to Fig.3, the detection is much more accurate.

B. Forward Kinematics

To solve for the end-effector of the RX-200 arm with respect to its joint angles, we use the Denavit-Hartenberg (D-H) to describe the forward kinematics of the arm. The D-H parameters provide a systemic method of defining relative transformations between joints using only an axis of translation and an axis of rotation - typically the x-axis and z-axis, respectively. Unlike a full homogeneous transformation matrix, which can account for arbitrary rotations and translations, the D-H parameters are more restrictive, being parameterized by only four values:

$$[\theta, d, a, \alpha]$$

where θ is the joint angle, representing a rotation about the z-axis, d is the link offset, representing a translation along the z-axis, a is the link length, representing a translation along the x-axis, and α is the joint twist, representing a rotation about the x-axis.

As a result, the D-H convention allows for the expression of a large 'subset' of all possible homogeneous transforms using only these four parameters, providing both simplicity and computational efficiency. This approach was originally pioneered during an era in which computational resources were limited, especially for complex robotics systems.

We use the D-H convention to define and build the forward kinematics transformations. Parameters are assigned for each joint in order to map the transformations corresponding to the geometry of the arm. Due to the geometry of the RX-200, an offset angle β for is introduced for joint frames 2 and 3 in order to account for the unique structure of the RX-200. This offset angle is subtracted from the joint angle of θ_2

Furthermore, joint frame 4 (parameterized by $[\theta_4, d_4, a_4, \alpha_4]$) is offset from its actual physical location due to the limitations in directly transforming between frame 3 and frame 4 while adhering to the D-H conventions. As a result, joint frames 3 and 4 are effectively coincident, and the transformation to the end-effector frame is adjusted accordingly in order to maintain geometric fidelity.

C. Inverse Kinematics

We solve for the configuration of our arm using the geometric implementation of inverse kinematics. Typically, a 5-DOF robotic arm requires numerical approaches to solve IK. However, due to the configuration of our experimental arm, where the rotation axes of joints 2, 3, and 4 are parallel, we could directly calculate the first joint angle using the x and y coordinates of the end-effector in the world frame. Knowing the end-effector's position and its approach angle relative to the horizontal plane allows us to determine the position of joint 4. This reduces the IK of the remaining joints to a 2-link RR problem, for which we consider only the "elbow-up" configuration. By knowing the lengths of link 2 (l_2) and link 3 (l_3), and calculating the arcsin value $\theta_4 = \arcsin\left(-\frac{x}{y}\right)$ of joint 4 from the frame coordinates, we can proceed with the calculations. The z-coordinate and the projection of the arm length in the x-y plane are also considered. To find θ_3, we apply the law of cosines:

$$\theta_3 = \arccos\left(\frac{l_2^2 + l_3^2 - \text{third side of the triangle}^2}{2 l_2 l_3}\right)$$

where the "third side of the triangle" represents the distance between the joint positions. After calculating θ_3, we use the arctangent function to calculate θ_2 as follows:

$$\theta_2 = \arctan\left(\frac{l_3 \sin(\theta_3)}{l_3 \cos(\theta_3) + l_2}\right)$$

The calculation of the length of the third side of the triangle is based on the height of the end-effector (z) and the projection of the joint4's position onto the x-y plane in the world coordinates. It involves multiplying the cosine of ψ by l_5, then subtracting this from the end-effector's position. The resulting value is divided by the cosine and sine of θ_1 to obtain the lengths in the x and y directions within the world coordinates. By calculating the square root of the sum of the squares of these two lengths, the projection length, which corresponds to the projection length in the x-y plane, further calculation of the third side length is the square root of the sum of the squares of this length and z squared. This approach provides the necessary joint angles for the inverse kinematics calculations.

III. Results

A. Checkpoint 1

Camera Calibration

1) The average intrinsic matrix is:

$$\mathbf{K}_{average} = \begin{bmatrix} 894.735 & 0 & 660.250 \\ 0 & 894.699 & 384.114 \\ 0 & 0 & 1 \end{bmatrix}$$

The factory intrinsic matrix is

$$\mathbf{K}_{factory} = \begin{bmatrix} 896.86 & 0 & 660.523 \\ 0 & 897.203 & 381.419 \\ 0 & 0 & 1 \end{bmatrix}$$

The average intrinsic matrix is very close to the factory calibrated one. In practice, we prefer to use the factory intrinsic matrix since their calibration process is more rigorous and sometimes our calibration could lead to totally different value if the calibration is not properly implemented. The error of the calibration may comes from lens distortion, reprojection error, etc.

2) The extrinsic matrix is

$$\mathbf{M}_{extrinsic} = \begin{bmatrix} 1 & 0 & 0 & 32 \\ 0 & \cos\theta & -\sin\theta & 285 \\ 0 & \sin\theta & \cos\theta & 1000 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \theta = 174^\circ$$

3) Teach and repeat

In the teach phase of teach and repeat, we record each waypoint and gripper status and store them in two separate lists in preparation for the subsequent setposition function. In the repeat experiment of teach and repeat, we can repeat a maximum of 4 times, the rosbag figure plot is shown in the figure below. It can be observed that with each repetition, the motor of joint 1 will produce a slight error toward the plane, causing the final end effector position to gradually lower, ultimately resulting in the inability to pick up the block.

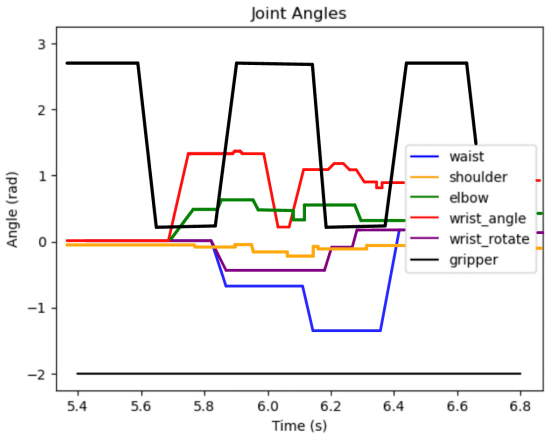


Fig. 5. Teaching and repeat Ros2bag playback

B. Checkpoint 2

Apriltag Calibration

1) Since we use auto-calibration routine using Apriltags, there is no depth calibration function. The extrinsic matrix by Apriltags calibration is

$$\mathbf{M}_{extrinsic} = \begin{bmatrix} 0.999 & -0.0048 & -0.016 & 35.55 \\ -0.004 & -0.997 & 0.082 & 167.44 \\ -0.016 & -0.082 & -0.996 & 1009.56 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Compared to the hand-measured one, it is quite close in x and z axis, but deviate in y axis, the error may be the camera position is moved a little bit due to small vibration and we also manually changed its position to do some testing.

2) We use the following procedure to transform [u,v,d] into world frame.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = d \cdot \mathbf{K}_{intrinsic}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \mathbf{M}_{extrinsic}^{-1} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

3) To verify the calibration is correct after the function updates the extrinsic matrix, we place 4 blocks at the corners of the workspace and measure. This allows us to compare the computed world coordinates with the actual positions. We then move the blocks slightly inward and repeat the process. However, we observed that the auto-calibration is less accurate along the z-axis, with inconsistencies across the workspace. To enhance the calibration, we manually apply a rotation matrix along the x-axis and fine-tune the rotation angle to achieve more consistent z-axis values. Some results are shown in the following Table II.

Blocks	Computed Position	Actual Position
Up-Left	(-305, 328, 39)	(-300, 325, 37)
Down-Left	(-307, -82, 33)	(-300, -75, 37)
Down-Right	(310, -82, 35)	(300, -75, 37)
Up-Right	(307, 330, 32)	(300, 325, 37)

TABLE I
POSITION OF BLOCKS WITH CALIBRATED EXTRINSIC MATRIX

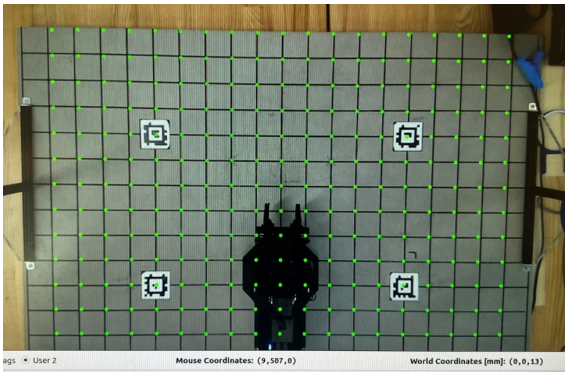


Fig. 6. grid point project

Validate Calibration & Homography

1) Fig.6 shows the grid point projection, indicating that our calibration is fairly accurate within the region containing the Apriltags. However, areas farther from the Apriltags exhibit some distortion, likely caused by lens distortion.

2) We use the center of the 4 Apriltags as the source points to calculate the perspective transformation since we could easily get their position. The homography matrix is

$$\mathbf{H} = \begin{bmatrix} 0.989 & -0.064 & 21.621 \\ 0.013 & 0.941 & 49.135 \\ 2.146 \times 10^{-5} & -8.977 \times 10^{-5} & 1 \end{bmatrix}$$

Forward Kinematics

1) Fig.7 shows how we computed the forward kinematics. Our DH table is as follows: 2) After finished

n	θ_n	d_n	a_n	α_n
1	$\pi/2 + \theta_1^*$	103.91	$-\pi/2$	0
2	$-\beta + \theta_2^*$	0	0	205.73
3	$\beta + \theta_3^*$	0	0	200
4	$-\pi/2 + \theta_4^*$	0	$-\pi/2$	0
5	θ_5^*	174.15	0	0

TABLE II
DENAVIT-HARTENBERG (D-H) TABLE

the forward kinematics function, we just manually move the end-effect to a position such as (0, 175, 100), (300, -75, 30), (-300, -75, 50), (300, 325, 0) and check the

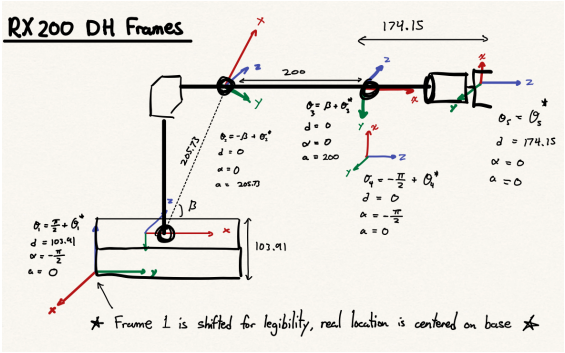


Fig. 7. Detailed procedure to compute forward kinematics

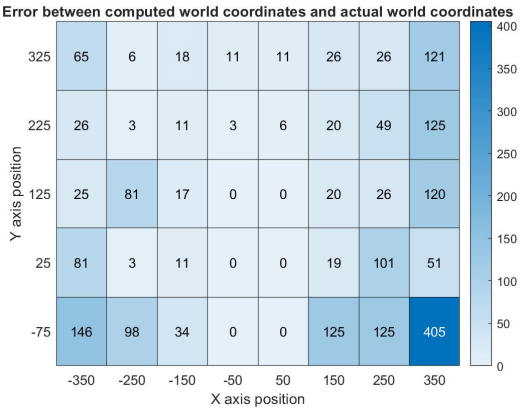


Fig. 8. The heatmap of the accuracy of position detection

output of the computed world coordinates. The actual corresponding computed coordinates are (3.28, 177.93, 103.74), (299.2, -79.52, 29.43), (-303.31, -71.54, 48.21), (306.46, 316.01, 5.25), and the actual error are within 9 mm.

C. Checkpoint 3

Block Detector

1) Fig.4 shows the result of the block detection and it could successfully detect the shape, size, color, position and orientation of the objects. We tested for different square blocks and some distractor object with different color, and the detection is for color is pretty accurate to distinguish from red to purple. And the accuracy of the position detection could be checked in Fig.8. To be noted, the zeros in the middle indicates the position of the arm.

Inverse Kinematics

Figure 6 presents the schematic diagram. On the left side, the top view of the first joint is displayed, while the right side shows the side view of all the joints and links. Notably, the direction of each joint is color-coded, with red, green, and blue representing the x, y, and z axes, respectively.

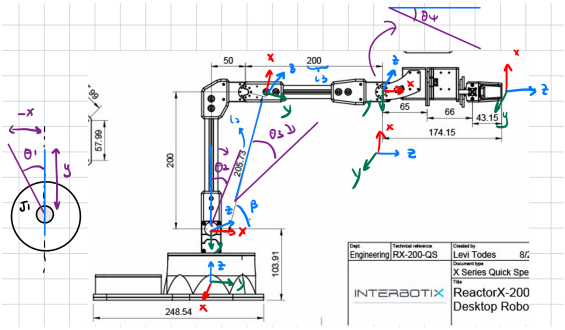


Fig. 9. FK and IK schematic

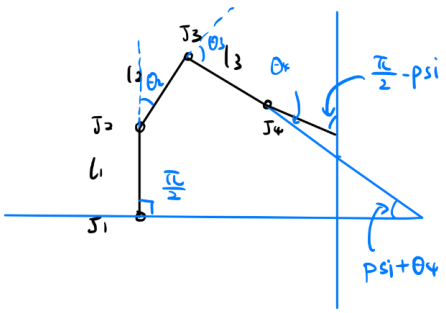


Fig. 10. Angle calculation for theta 4

2) The target position is denoted as x_t, y_t, and z_t. First, to calculate the angle for joint 1, since the rotation angle and the positive x-axis definition are opposite, θ_1 can be calculated using the following formula:

$$\theta_1 = \arctan\left(\frac{-x_t}{y_t}\right)$$

(as shown in line 236 of the appendix). Next, we determine the world coordinates of joint 4 by subtracting the length of link 4 projected onto the x-y plane. The coordinates of joint 4 are denoted as x_4, y_4, and z_4 (lines 239-249). The calculation of x_4 is as follows:

$$x_4 = \begin{cases} x_t - 174.15 \cdot \cos(\psi) \cdot \sin(\theta_1) & \text{if } x_t > 0 \\ x_t + 174.15 \cdot \cos(\psi) \cdot \sin(\theta_1) & \text{if } x_t < 0 \end{cases}$$

This accounts for subtracting the x-direction projection in both positive and negative directions, adjusting x_4 accordingly. The calculation for y_4 is similar:

$$y_4 = \begin{cases} y_t - 174.15 \cdot \cos(\psi) \cdot \cos(\theta_1) & \text{if } y_t > 0 \\ y_t + 174.15 \cdot \cos(\psi) \cdot \cos(\theta_1) & \text{if } y_t < 0 \end{cases}$$

For z_4, the calculation is as follows:

$$z_4 = z_t + 174.15 \cdot \sin(\psi) - 103.91$$

This adds the z-axis projection of link 5 to z_t to calculate z_4. There is no need to account for $z_t < 0$ since in our project, the target z_t is always greater than zero. Additionally, the reason for subtracting link 1 length, is to suit the RR model for further calculations.

We also calculate $\bar{y}$ and $\bar{x}$ for the RR link model calculations. $\bar{y}$ corresponds to the z-coordinate of joint 4, which is directly equal to z_4 (line 249). On the other hand, $\bar{x}$ represents the projection of the distance from the origin to joint 4 in the x-y plane (lines 250-253). The calculation of $\bar{x}_4$ is as follows:

$$\bar{x}_4 = \begin{cases} \sqrt{x_4^2 + y_4^2} & \text{if } y > 0 \\ -\sqrt{x_4^2 + y_4^2} & \text{if } y < 0 \end{cases}$$

This defines $\bar{x}_4$ based on the sign of y, ensuring the correct projection in the x-y plane.

The calculation for θ_3 and θ_2 follows the methodology described earlier, we calculate θ_3 using the cosine theorem. Once θ_3 is obtained, it allows us to construct a new triangle. Using this triangle, we apply the arctangent function to determine the value of θ_2. (lines 257-260). The formula for θ_3 is:

$$\theta_3 = \arccos\left(\frac{\bar{y}_4^2 + \bar{x}_4^2 - (205.73)^2 - (200)^2}{2 \cdot 205.73 \cdot 200}\right)$$

The formula for θ_2 is:

$$\theta_2 = \arctan 2(\bar{x}_4, \bar{y}_4) - \arctan 2(200 \sin(\theta_3), 205.73 + 200 \cos(\theta_3))$$

As shown in the figure, the sum of the internal angles is 2π. Two of these internal angles are $\pi - \theta_2$ and $\pi - \theta_3$. Since link 1 is always perpendicular to the x-y plane, the third angle is $\frac{\pi}{2}$. The last internal angle is related to ψ and θ_4, and can be calculated as:

$$\text{Fourth angle} = \frac{\pi}{2} - \left(\frac{\pi}{2} - \psi - \theta_4\right) = \psi + \theta_4$$

The calculation for θ_4 is based on the sum of the internal angles being 2π. The internal angles consist of $\pi - \theta_3$, $\pi - \theta_2$, $\frac{\pi}{2}$ (since link 1 is perpendicular to the x-y plane), and $\psi + \theta_4$. Therefore, the equation is:

$$2\pi = (\pi - \theta_3) + (\pi - \theta_2) + \frac{\pi}{2} + (\psi + \theta_4)$$

Rearranging this equation, we solve for θ_4:

$$\theta_4 = 2\pi - \left((\pi - \theta_3) + (\pi - \theta_2) + \frac{\pi}{2} + \psi\right)$$

we can derive the relationship between θ_4, θ_2, and θ_3. Therefore, θ_4 (as shown in line 262) is calculated as:

$$\theta_4 = \psi + \frac{\pi}{2} - \theta_2 - \theta_3$$

The next step is to determine the rotation of the end-effector. If the end-effector is vertically approaching the target, we directly use the rotational angle combined with θ_1 to ensure that the robotic arm remains aligned with the direction of the target cubes. In the case of horizontal approaches, we set the rotation to zero to ensure a horizontal grasp by the end-effector (as described in lines 265-268). The final step is to adjust the angles for the initial setup, converting them into joint angles. For θ_2, we subtract $\arctan 2\left(\frac{50}{200}\right)$, and for

θ_3, we subtract $\arctan 2\left(\frac{200}{50}\right)$. This accounts for the geometric relationship compared to the initial positions (as described in lines 272-273).

Click to Grab & Drop

In the algorithm used for the "click and grab" operation, we first obtain the world coordinates by capturing the last click attributes from the camera class. Next, we use three specially designed functions to carry out the operation. The first function is called `grabBlock`, which is responsible for grabbing the block. This process involves two waypoints to avoid collisions with the cube during the grabbing motion.

Initially, we calculate the `waypoint_high` by setting $z_t + 50$ (line 215) to ensure the gripper is positioned above the cube. Then, we calculate the `waypoint_low` as $z_t - 18$ (line 217). The reason for subtracting 18 is that the target point, as identified by our computer vision algorithm, is on the top of the cube. To grip the cube properly, we lower the gripper slightly to ensure it surrounds the cube.

Next, the IK function is used to generate the joint angles, with the settings for vertical approach and the rotational angle provided by the computer vision system (lines 219-220). An exception is made if the distance to the cube is greater than 400. In this case, we perform a horizontal approach, where the `waypoint_high` changes to $(x_t - 50, y_t - 50, z_t + 50)$ and the `waypoint_low` changes to $(x_t, y_t, z_t - 20)$. Additionally, the end-effector's approach angle is set to 0 in the IK function (lines 234-239).

The next function is `releaseBlock`, which handles three different conditions. In each condition, we define three waypoints: `position_low`, `position_high`, and `position_prepare`. Similar to `grabBlock`, the robot first moves to `position_prepare`, then to `position_high`, and finally to `position_low`.

The first condition occurs when the target height is low (less than 150) and the distance is within 325. In this case, the vertical approach is used (lines 253-264). The other two conditions apply when the height is still less than 150 but the distance is greater than 325, and when the height is greater than 150. For both of these cases, a horizontal approach is used (lines 266-289).

The last function is `rollback`, which prevents the next grabbing movement from colliding with the previously released block (lines 347-364). The conditions are as follows: if the height is less than 150 and the distance is less than 350, we move to a higher position $(x_t, y_t, z_t + 110)$, with IK solving for a $\frac{\pi}{2}$ vertical approach. If the distance is greater than 350, we move to a higher position $(x_t, y_t, z_t + 150)$, with IK using a horizontal geometric approach. The final condition applies when the height is greater than 150, where we move to $(x_t, y_t, z_t + 100)$ and use a horizontal approach.

IV. Discussion

A. Visual Detection

Our visual detection system is quite robust, offering highly accurate color and positional detection even under changing lighting conditions. The only challenge arises in detecting the shapes of objects. The cv2.approxPolyDP() function sometimes struggles with small squares, occasionally detecting 6 vertices instead of 4 or 5. Relaxing the conditions for detecting squares risks misclassifying other shapes, such as circles, as squares. However, after fine-tuning the height-to-width ratio, the system reliably distinguishes between squares and rectangles.

We've considered two potential solutions to improve shape detection. The first is to upgrade to a more precise depth camera, which would be straightforward but comes with higher costs. The second approach is to perform shape detection in the RGB image as well, and then cross-check the results with the depth image. An object would be classified as a square only if both detection methods agree. While this method is more complex and time-consuming, it could increase accuracy, though some limitations may still persist.

B. Forward Kinematics

While the Denavit-Hartenberg (D-H) convention was effective in modelling the RX-200 arm's joint configurations, there are inherent limitations in its four-parameter approach. Firstly, while the D-H parameters are computationally efficient, said efficiency comes at a cost: the parameters are constrained by their reliance on ordered z-axis and x-axis transformations, which imposes heavy restrictions on flexibility in representing certain robot geometries. This is evident in the introduction of a "virtual joint" for the RX-200 wrist (joint frame 4) to accommodate for the inability to directly transform between the elbow and wrist joints, as well as the introduction of the offset angle β between joint frame 2 & 3 for similar purposes.

By incorporating the use of full homogeneous transforms and quaternions, the conceptual process for building the forward kinematics of the RX-200 could be greatly simplified. Homogeneous transforms allow for arbitrary rotations and translations in three-dimensional space, completely eliminating the need for workarounds such as virtual joints or offset angles.

C. Inverse Kinematics

When using IK to solve for the target point, we did not impose limits on the joint angles. As a result, when the position falls outside the working range, the IK function produces NaN values, which aborts the entire movement

process. Additionally, we did not apply numerical methods, such as the Newton-Raphson method, to solve for the IK. For more general applications, if the rotational axes of joints 2, 3, and 4 are not parallel, the geometric solution becomes limited and cannot provide closed-form solutions.

By incorporating numerical methods and comparing them with geometric solutions, while setting accurate joint angle limits and filtering the input based on the workspace constraints, the entire IK process would become more robust.

V. Conclusion

In this lab, we successfully demonstrated the practical application of forward and inverse kinematics, combined with computer vision, to achieve autonomous manipulation tasks using the RX-200 robotic arm. By integrating the Intel Realsense LiDAR Camera L515 for object detection and implementing both kinematic models, the system was able to sort, stack, and manipulate objects with high accuracy. Our work highlights the importance of precise camera calibration and robust kinematic solutions in ensuring reliable robotic performance.

The challenges encountered, such as minor misalignment issues during object detection and the limitations imposed by the Denavit-Hartenberg parameters in the forward kinematics, were addressed through both software refinements and potential future improvements like camera upgrades or hybrid detection techniques. Additionally, the exploration of geometric inverse kinematics provided an efficient solution for our 5-DOF arm, but future implementations would benefit from the integration of numerical methods for greater versatility.

Overall, the project reinforced key robotic concepts and provided valuable insights into real-world robotic control systems. The methodologies applied here can be extended to more complex systems, and the lessons learned will guide future developments in autonomous robotic manipulation.

References

VI. Appendices

```python
def IK_geometric(tgt_pose, psi_angle, end_effector_rot):
    y_t = tgt_pose[1]
    z_t = tgt_pose[2]
    psi = psi_angle

    # Solve for joint1 (rotational base)
    theta_1 = np.arctan2(-x_t,y_t)

    # Use trig to solve for joint 4
    if (x_t>=0):
        x4 = x_t - 174.15*np.cos(psi)*np.sin(theta_1)
    else:
        x4 = x_t + 174.15*np.cos(psi)*np.sin(theta_1)
    if (y_t>=0):
        y4 = y_t - 174.15*np.cos(psi)*np.cos(theta_1)
    else:
        y4 = y_t + 174.15*np.cos(psi)*np.cos(theta_1)
    z4 = z_t + 174.15*np.sin(psi) - 103.91
    # Shift coords into 2d plane so we can use lecture slides
    y4_bar = z4
    if y4 >= 0:
        x4_bar = np.sqrt(x4**2+y4**2)
    else:
        x4_bar = -np.sqrt(x4**2+y4**2)


    # Formula for joint 3 (theta2 in slides - law of cosines)
    theta_3 = np.arccos((y4_bar**2 + x4_bar**2 - (205.73)**2 - (200)**2) / (2*205.73 * 200))

    # Formula for joint 2 (theta1 in slides)
    theta_2 = np.arctan2(x4_bar, y4_bar) - np.arctan2(200*np.sin(theta_3), 205.73 + 200*np.cos(theta_3))

    theta_4 = psi + (1/2) * np.pi - theta_2 - theta_3

    #
    if psi_angle < np.pi/4:
        theta_5 = 0
    else:
        theta_5 = end_effector_rot + theta_1


    # Account for offset robot geometry
    theta_2 = theta_2 - np.arctan2(50, 200)
    theta_3 = theta_3 - np.arctan2(200, 50)

    joint_angles = np.array([theta_1,theta_2,theta_3,theta_4,theta_5])
    return joint_angles
```

Fig. 11. Appendix for IK kinematics

```python
def grabBlock(self, blockPose):
    x_t = blockPose[0]
    y_t = blockPose[1]
    z_t = blockPose[2]
    angle = blockPose[3]
    #test if 90 degrees of approaching angle of end effector is suitable

    if(x_t**2 + y_t**2) <400**2:
    # Get to a positon that is bit higher than block center
        posHigh = np.array([x_t, y_t, z_t + 50])
    # Grab the block at a positon that is lower than the block center
        posLow = np.array([x_t, y_t, z_t - 18])
    # Go to high pos -> grab block -> go to high pos
        waypoint_1 = IK_geometric(posHigh, np.pi/2, angle)
        waypoint_2 = IK_geometric(posLow, np.pi/2, angle)
        waypoints = np.vstack([waypoint_1, waypoint_2])
    #if larger than this range:
    # elif (325**2) <= (x_t**2 + y_t**2) <= 375**2:
    #     print("far")
    #     posLow = np.array([x_t, y_t, z_t - 18])
    #     if x_t > 0:
    #         posHigh = np.array([x_t-50, y_t-50, z_t + 150])
    #     else:
    #         posHigh = np.array([x_t+50, y_t-50, z_t + 50])
    #     waypoint_1 = IK_geometric(posHigh, 0, angle)
    #     waypoint_2 = IK_geometric(posLow, np.pi/2, angle)
    #     waypoints = np.vstack([waypoint_1, waypoint_2])
    else:
        print("too far")
        posHigh = np.array([x_t-50, y_t-50, z_t+50 ])
        posLow = np.array([x_t, y_t, z_t - 20])
        waypoint_1 = IK_geometric(posHigh, 0, 0)
        waypoint_2 = IK_geometric(posLow, 0, 0)
        waypoints = np.vstack([waypoint_1, waypoint_2])
```

Fig. 12. Appendix for grabBlock

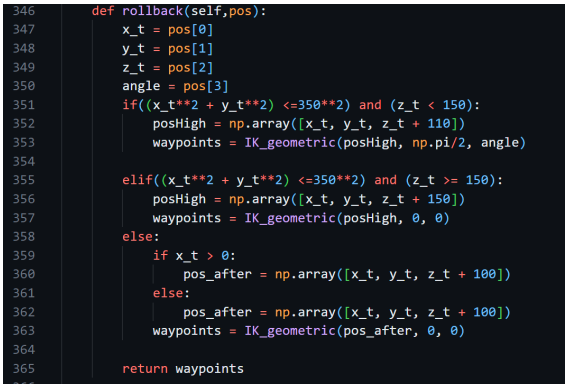
```
    def releaseBlock(self, releasePostion):
        x_t = releasePostion[0]
        y_t = releasePostion[1]
        z_t = releasePostion[2]
        angle = releasePostion[3]
        print(releasePostion)
        #if the releasing position less than 15cm
        print((z_t < 150) and ((x_t**2+y_t**2)<(325**2)))
        if (z_t < 150) and ((x_t**2+y_t**2)<(325**2)):
        # Get to a positon that is bit higher than release position
            pos_prepare = np.array([0, 125, 100])
            posHigh = np.array([x_t, y_t, z_t + 80])
            posLow = np.array([x_t, y_t, z_t + 19])

        # Go to high pos -> grab block -> go to high pos
            waypoint_0 = IK_geometric(pos_prepare, np.pi/2, angle)
            waypoint_1 = IK_geometric(posHigh, np.pi/2, angle)
            waypoint_2 = IK_geometric(posLow, np.pi/2, angle)
            waypoints = np.vstack([waypoint_0,waypoint_1, waypoint_2])
            print("if case")
        # if position is higher than 15 cm, horizontal approach would be mades
        elif (z_t < 150) and ((x_t**2+y_t**2)>(325**2)):
        # Get to a positon that is bit higher than release position
            pos_prepare = np.array([0, 125, 100])
            posHigh = np.array([x_t, y_t, z_t + 200])
            posLow = np.array([x_t, y_t, z_t + 19])

        # Go to high pos -> grab block -> go to high pos
            waypoint_0 = IK_geometric(pos_prepare, np.pi/2, angle)
            waypoint_1 = IK_geometric(posHigh, 0, angle)
            waypoint_2 = IK_geometric(posLow, 0, angle)
            waypoints = np.vstack([waypoint_0,waypoint_1, waypoint_2])
        else:
            #this is the position on Z-asix
            pos_prepare = np.array([0, 100,350])
            if x_t > 0:
                pos_intermidate = np.array([x_t-30, y_t-30, z_t + 20])
            else:
                pos_intermidate = np.array([x_t+30, y_t-30, z_t + 40])
            pos_goal = np.array([x_t, y_t, z_t+20])

            waypoint_3 = IK_geometric(pos_prepare, np.pi/3, 0)
            waypoint_4 = IK_geometric(pos_intermidate, 0, 0)
            waypoint_5 = IK_geometric(pos_goal, 0, 0)
            waypoints = np.vstack([waypoint_3, waypoint_4,waypoint_5])
```

Fig. 13. Appendix for releaseBlock



```
    def rollback(self,pos):
        x_t = pos[0]
        y_t = pos[1]
        z_t = pos[2]
        angle = pos[3]
        if((x_t**2 + y_t**2) <=350**2) and (z_t < 150):
            posHigh = np.array([x_t, y_t, z_t + 110])
            waypoints = IK_geometric(posHigh, np.pi/2, angle)

        elif((x_t**2 + y_t**2) <=350**2) and (z_t >= 150):
            posHigh = np.array([x_t, y_t, z_t + 150])
            waypoints = IK_geometric(posHigh, 0, 0)
        else:
            if x_t > 0:
                pos_after = np.array([x_t, y_t, z_t + 100])
            else:
                pos_after = np.array([x_t, y_t, z_t + 100])
            waypoints = IK_geometric(pos_after, 0, 0)

        return waypoints
```

Fig. 14. Appendix for rollBack