ArmLab: ROB550 Project Report

Leo Bringer, Hannah Ho, Jiangbo Yu
lbringer@umich.edu, hdho@umich.edu, jumboyu@umich.edu

***Abstract*—This project aims to create a controller that is able to control a 5 DOF robot arm in order to perform certain tasks autonomously, using sensors to determine the location of arm and objects surrounding it. The end goal being to manipulate cubes across the workspace.**

I. Introduction

The armlab project is about controling the 5 degree of freedom ReactorX-200 robot arm both autonomously and teleoperated. The image below is the engineering drawing of the robot.

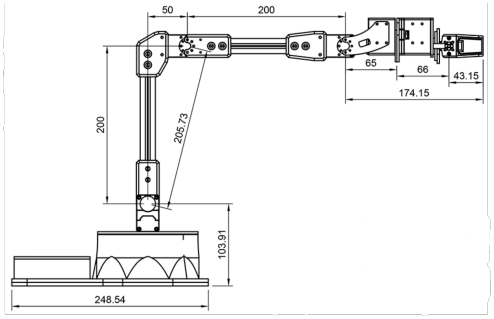

Fig. 1. Engineering drawing of the arm provided by Trossen Robotics at https://www.trossenrobotics.com/reactorx-200-robot-arm.aspx#drawings

Using the sensors within the motors for position as well as an RGB-D camera, we determine the position of the robot as well as objects within the workspace. Using block detection, forward kinematics, inverse kinematics and path planning, we are to perform tasks autonomously during an end-of-project competition. The competition entails sensing, manipulating and stacking blocks of various colors, sizes, and shapes.

II. Methodology

A. Computer Vision

1) Camera Calibration: To enable our robot to perceive and interact with the environment effectively we used an RGB-D camera, but to allow an accurate alignment of visual data with the robot's physical workspace and in order to get workspace coordinates from the camera, we performed a camera calibration. The camera calibration is twofold, the intrinsic and extrinsic part:

The intrinsic calibration allows the mapping between the pixel coordinates and the camera coordinates in the image frame. The method we used to find the intrinsic matrix is commonly called Zhang's method. The idea is to capture images of a planar calibration pattern with known dimensions (a checkerboard in our case), place it in different positions and orientations within the camera's field of view. Then, a mathematical model relates the 3D coordinates of the calibration pattern in the real world to their 2D coordinates in the image plane and then solve a non-linear optimization problem to find the best-fitting camera parameters that minimize the difference between the observed 2D image points and the 3D model points projected into the image space. The output of each calibration process is a 3x3 K matrix that contains all information about the intrinsic parameters.

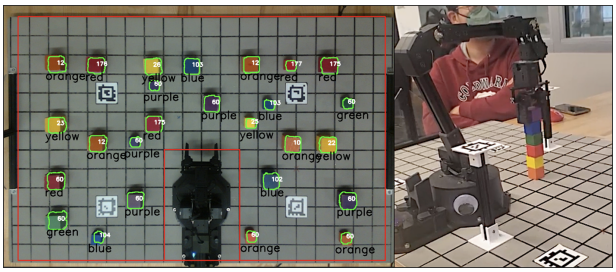

Fig. 2. Visualisation of the Block Detector - Precise Pick & Place during the competition

The extrinsic calibration allows to get the rotation and translation of the camera with respect to some world coordinate system. In our case the extrinsic calibration procedure has been done at three different levels: First, a naive version with rough approximations from physical measurements of the lab apparatus, then an improved version leveraging the camera's accelerometer (Figure 3), and finally we implemented an auto-calibration routine using AprilTags. For this final and most accurate version we use 6 different AprilTags located at each corner of the workspace and at difference heights. For each marker we have access to its center and four corners, which makes a total of 35 points. Then we solve a PnP system (Perspective-n-Point) of equations that minimize the reprojection error from 3D-2D point correspondences. The 3D points are the fixed x,y,z coordinates of each of the Apriltags in the world reference and the associated 2D points are their u,v coordinates recognized by the camera. The solution of this PnP system is a

rotation vector and translation vector that leads to a 4x4 extrinsic matrix using the Rodrigues Formula.

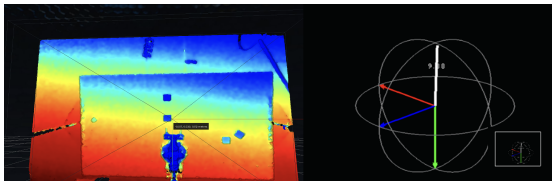

Fig. 3. realsense-viewer program to leverage the camera's accelerometer

2) Workspace Reconstruction:

(i) Homography Transform:
An homography transform, also known as a projective transform, is a geometric transformation that maps points from one plane to another while preserving collinearity and ratios of distances. In our specific case, this allowed us to rectify the camera's viewpoint, transforming it from a tilted angle to a top-down perspective (bird's-eye view). To perform this homography transform we used an OpenCV function that finds the closest homography matrix between a set of points in the original pixel image and their desired location in the output pixel image. The extrinsic and intrinsic matrices previously determined led us to the position of the corners of the grid in the pixel image from their position in the world reference coordinates. We used these 4 corner grid points as the set of points in the original pixel image and placed on the corner of the output image (Figure 4).

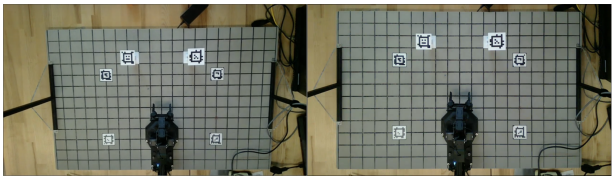

Fig. 4. Before & after the perspective transform for the bird's-eye view

(ii) From pixel to 3D coordinates in the reconstructed workspace:
One important feature that we implemented is the display of the world coordinates as we track the mouse location hovering over the RGB video image on bird's-eye view. To do that, we kept track of the (u_0, v_0) coordinates of the mouse on the pixel image (before perspective transform) and access the associated depth value d, on the depth image from the depth sensor (not been modified by the previously stated homography transform). To obtain the (u_0, v_0) coordinates on the original image we keep track of the (u, v) coordinates on the bird's-eye view and apply the inverse of the projective transform H_p:

$$\begin{bmatrix} u_0 \\ v_0 \\ 1 \end{bmatrix} = H_p^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{1}$$

Then from this we leverage the intrinsic and extrinsic matrix (K and H) to obtain the world coordinates of the associated pixel value of the mouse location:

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = Z_C \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \cdot K^{-1} \begin{bmatrix} u_0 \\ v_0 \\ 1 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = H^{-1} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \tag{3}$$

Finally to verify that our calibration was correct, we placed our mouse on positions on the RGB Video corresponding to known world coordinates such as the Apriltags or over the x and y-axis of the grid and compared the obtained values with the actual values.

3) Block Detection: To allow the autonomous pick and place of colored blocks on a grid, the development of the block detector holds paramount importance. The block detector's primary objective is to identify the characteristics of each block, including its shape, size, color, and orientation. Our approach to achieving this can be broken down into two key components.

First, we use the depth image to detect the geometric properties of each block. This involves creating a mask for each block using upper and lower depth bounds, leveraging our knowledge of the approximate height of each block above the grid. This step allows us to precisely determine the block's shape and orientation in the 3D environment. The second part of our methodology employs the RGB image to extract the color information of the detected blocks. By applying these masks on the RGB image, we can effectively determine the color of each block within the workspace.

We used OpenCV for geometry determination and conducted extensive testing of our block detection implementation using sample RGB and depth images collected from the robot's workspace. Our code incorporates a range of strategies to enhance the robustness of the detector and minimize false positive detections. These strategies include filtering the contours based on area and aspect ratio, ensuring that only well-defined blocks are

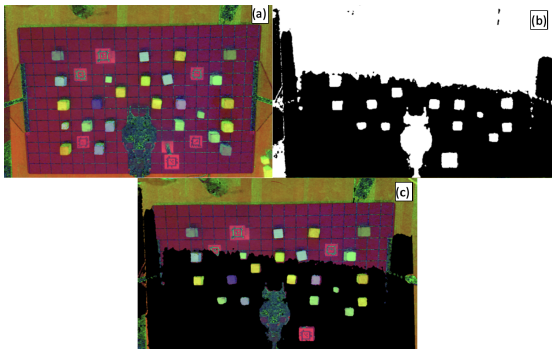

Fig. 5. (a) Original HSV image - (b) Depth block contours using distance bounds - (c) retrieved color from depth masks

considered. The first filter removes the contours with an area greater than a maximum value and the second one constrains the height and width of a block. This filtering methods help to avoid confusions with other objects than the blocks lying in the workspace. Additionally, we apply a mask to exclude regions outside the robot's workspace and effectively filter out the arm's presence (Figure 6).

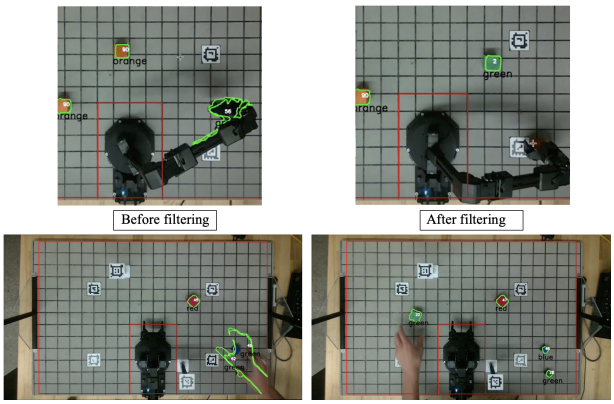

Fig. 6. Before & after using filtering methods on the contours

Another import detail for the block detection is the depth calibration. As you can see on Figure 5, the camera is titled and the homography transform does not correct the depth slope. Hence, we used an offset in order to leverage depths bounds to create the contours for the block detection. We thus added a button on the GUI called "Depth Calibration", which records the current value of the depth and subtracts this depth offset to each depth frame to obtain a plain grid calibrated around the XY plane. This means that the workspace needs to be cleared while performing the depth calibration.

Finally, the last important issue we tackled to obtain our fine-tuned block detector, was confusion errors between the colors retrieved from each of the filtered contours. Our initial method was using the RGB (Red-Green-Blue) color space convention. We used a recorded

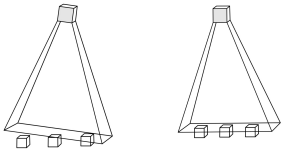

Fig. 7. Depth calibration by subtracting an inital offset for the creation of contours using depth bounds from the depth sensor

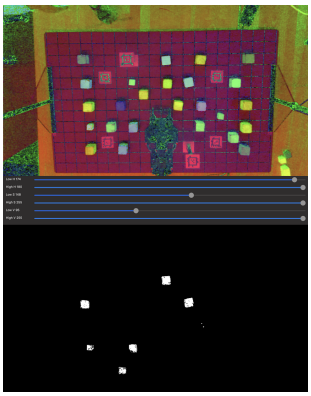

Fig. 8. Determining HSV range for contour color retrieving (e.g. for a blue block)

sample image of our workspace to determine predefined values of each expected block color in the RGB color space. Then for each filtered contour we calculated the mean RGB value of the pixels in the contour and determined the color retrieved by taking the distance (l2-norm) from this mean value to each color. The output being the color corresponding to the shortest distance value from the mean. However, if this process functioned for most colors, for very close colors in terms of RGB value distance (like blue & green or orange & yellow) confusions were often made. To avoid those confusions, we used a different color space, HSR (Hue-Saturation-Value). Then, we determined ranges of HSV values corresponding to each color block also on a recorded sample image of our workspace (Figure 8). Then for each filtered contour we calculated the mean HSV value of the pixels in the contour and determined the retrieved color based on the ranges previously defined. We also tried with using the median or the mode (number that occurs most often in a data set) instead of the mean and after testing each method we found the best results using the median of the HSV value.

B. Robot Controls

To properly understand and control where the robot arm is in space, we determined the algorithms to determine the forward and inverse kinematics. Forward kinematics is the calculation to find where the end-effector is

given the angles of each joint in the arm. Inverse kinematics is the calculation to find the joint positions given a desired end-effector position and orientation. Figure 1 provided us with important measurements during our calculations.

1) Forward kinematic solution: Forward kinematics (FK) plays a vital part in determining the end-effector position. There are two ways to approach FK: Denavit-Hartenberg parameters (DH parameters) and Product of Exponentials (PoE), we decided to use the PoE approach because of its simplicity and versatility compared to the other approach. We also had issues with the DH Parameters we couldn't resolve in decent time. Using Fig. 1, we were able to construct our own drawing which includes the various fixed frames in the system as seen in figure 9. The frame at the base and the end-effector being the most important.

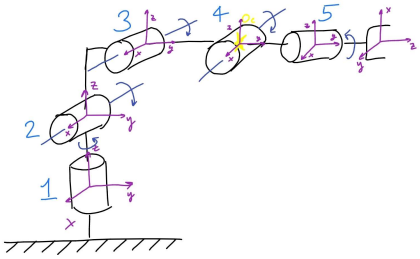

Fig. 9. Stick diagram of the arm robot including fixed frames

In order to test how accurate the forward kinematics are, we used the jaw to hold a AprilTag. Then we read the location of the AprilTag and cross check that with the location the end-effector should be.

2) Inverse kinematic solution: To automate the arm movement given a desired end-effector position, we used inverse kinematics (IK). We approached IK by using kinematic decoupling. We separated the arm into two groups: the joints before the wrist and after the wrist as seen in figure 10. The first group gives us the location, and the second gives us the orientation. This allows us to simplify the movement. Utilizing the laws of cosine, we found a variety of angles that were used to find the final joint angles θ_1 and θ_2.

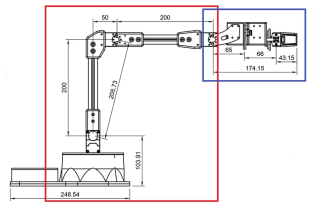

Fig. 10. Kinematic decoupling of the arm. Red is before the wrist, blue is after the wrist

We use FK to determine the position of the wrist given the desired orientation of the gripper and the end-effector position.

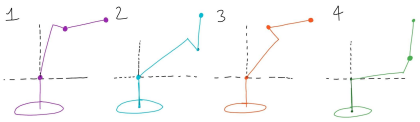

Fig. 11. Examples of each of the four possibilities of arm orientation. 1. Base at 0°, elbow up 2. Base at 0°, elbow down 3. Base at 180°, elbow up 4. Base at 180°, elbow down

Once the four orientations were determined, we calculated the angles of joints 1-3 for each orientation. Joints 4 and 5 were determined based on the desired orientation, taking into account the orientation of the wrist set by the first three angles.

C. Path planning

In order to tackle the complex environment in different task, we decided to employ some commonly-used path planning algorithm. Typically for robot arm, path-planning is done in high dimension space, thus making sample-based algorithms a great fit for this kind of task. In this case, we use RRT as our base algorithm for all the tasks.

1) base algorithm: There are basically two pathways to do path planning, either in Euclidean space or joint space. It is always more straight-forward to do path-planning in Euclidean space, but this method suffers from inaccuracy which comes from the frequent transformation between spaces using IK and FK. So, we focus on planning in the joint space and the pseudo code is shown in Algorithm 1.

Algorithm 1 RRT algorithm

Input: obstacles obs, starting point $start$, ending point end, step length $step$
Output: path from start to end $path$

$nodeList.append(start)$
while $dis > step$ **do**
  $n_{rand} \leftarrow RandomNode()$
  $n_{nearest} \leftarrow NearestNode(n_{rand})$
  $n_{new} \leftarrow Extend(n_{nearest}, n_{rand})$
  if $CheckCollision(n_{new})$ is False **then**
    $nodeList.append(n_{new})$
    $dis \leftarrow Norm(n_{new}, end)$
  else
    continue
$path \leftarrow GeneratePath(nodeList)$

The inputs are a list of obstacles, which we will get from the block detector, a starting point, which in most

cases is the current configuration of the robot arm, an ending point and the step length. Each Node on the tree contains a joint vector, its parent and its distance to the starting point.

$RandomNode()$ will generate a random node n_{rand} in joint space. In addition, to let algorithm converge faster, every time we generate a random node, we set it to the ending point by some probability, so that the search direction will be timely changed to where the ending point lives.

$NearestNode()$ will search in the entire tree for the node $n_{nearest}$ that has the shortest distance from the random node in joint space. $n_{nearest}$ will later become the parent of the new node in this iteration if it pass the collision check.

$Extend()$ will generate the new node n_{new} according to following equation:

$$v_{new} = v_{nearest} + \frac{v_{rand} - v_{nearest}}{\|v_{rand} - v_{nearest}\|} \cdot d_{step} \quad (4)$$

2) collision check: In all the tasks, items that will show up on the board or within the workspace of the robot arm only contains blocks and, in most cases, they are cubic blocks. Thus, it's reasonable to use cylinders to represent the obstacles in the environment. For cylinders, we use a line segment (p_a, p_b) and a radius r to specify it in our algorithm.

$CheckCollision()$ will take in two cylinders to check whether they have intersections. We first examine the minimum distance between the two line segments(p_{a1}, p_{b1}) and (p_{a2}, p_{b2}). Furthermore, to make calculation easier, we use capsules to represent the obstacle. Consequently, we only need to compare the sum of the radius of the two cylinder $r_1 + r_2$ and the distance we just got to examine whether two items are colliding or not. The introduction of capsule greatly reduces the complexity of collision check by eliminating special cases. An collision check example is shown in Fig.12

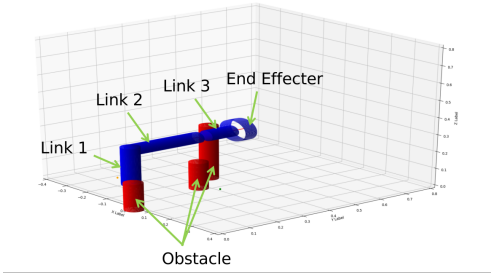

Fig. 12. Examples of a set of the cylinders that will be considered in collision check

We use FK and takes in the joint values from the node to generate a cylinder for each link of the robot. If all the links are not colliding with any of the obstacles in the obstacle list, then the collision check is passed.

3) Improvement on algorithm performance: RRT is actually a rather elementary and straight-forward method for path-planning and it suffers from non-optimal solution and potential failure in convergence within limited iterations. Much research and development has been done to improve its computational efficiency and its speed towards convergence. We modified our base algorithm into an RRT*-connect algorithm, as is shown in Algorithm 2.

Algorithm 2 RRT*-connect algorithm

Input: obstacles obs, starting point $start$, ending point end, step length $step$
Output: path from start to end $path$

$nodeListA.append(start)$ ▷ holds all a_i
$nodeListB.append(end)$ ▷ holds all b_i
while $dis > step$ **do**
  $a_{rand} \leftarrow RandomNode()$
  $a_{nearest} \leftarrow NearestNode(nodeListA, a_{rand})$
  $a_{new} \leftarrow Extend(a_{nearest}, a_{rand})$
  if $CheckCollision(a_{new})$ is False **then**
    $nodeListA.append(a_{new})$
    $AdjustTree(a_{new})$
  else
    continue
  $b_{nearest} \leftarrow NearestNode(nodeListB, a_{new})$
  $b_{new} \leftarrow Extend(b_{nearest}, a_{new})$
  if $CheckCollision(b_{new})$ is False **then**
    $nodeListB.append(b_{new})$
  else
    continue
  $dis \leftarrow Norm(a_{new}, b_{new})$
$path \leftarrow GeneratePath(nodeListA, nodeListB)$

Our algorithm actually evolves from RRT to RRT-connect and then to RRT*-connect. The first evolution mainly focused on adding another tree $nodeListB$ in the workspace which grows from the ending point. When any two of the nodes a_i, b_j on the two tree get closer than d_{step}, then we connect these two nodes and generate the path. The advantages of adding another tree starting from the end point are that it helps improve the situation where the ending point is surrounded by obstacles, and it can double the growing speed of the tree.

The second evolution is about how to generate at least a near optimal trajectory. In order to do this, two strategies are applied in function $AdjustTree()$.

$AdjustTree()$ will take in the newly-generated node a_{new} and try to adjust the structure of the tree. The first step is searching for nearby nodes $a_m, a_n, a_k, ...$ within a certain radius r_{thres} around a_{new}. Typically, $r_{thres} =$

$1.5d_{step}$. If their distance to the origin of the tree is smaller than that of $a_{new}.parent$, they will be made to become the new parent of a_{new}. The second step is to assign a_{new} to be the parent of the remaining nodes in this local area. Both these approaches will try to shorten the length of the final trajectory and avoid meaningless path points.

D. State machine

In state machine, our logic for different tasks are basically the same except for some changes to perform better in a specific task. Basically, we will update the list that contains all the information about the blocks in the workspace. We distinguish them by color and contour area and then clear the work space by moving all the blocks to specific locations to ensure enough space for performing the task.

In all tasks, a majority of time is spent in picking and placing, so it is very important to have a good precision when you grab the blocks, which comes from two techniques we use. The first one is that we use the angle of the contour we get from the block detector for each block, so the jaw is always aligned to the surface of the block. This technique can guarantee that when we approach the block and close the gripper, the block won't change a lot in position. The second technique is that we open and close the gripper twice to align the center of the block and the center of the gripper. To achieve this, we first go to the position of the block, close the gripper and re-open it. Then, we rotate the gripper by 90 degrees and close the gripper again. The first try of grabbing will align the center of the block in one direction and the second try will align both directions.

III. Results

A. Computer Vision

1) Intrinsic Calibration: After performing manual checkerboard calibration 5 times, we obtained 5 different intrinsic matrices. This is the average intrinsic matrix K_{Manual} from our checkerboard manual calibrations, and the factory calibration matrix provided by the camera constructor K_{Factory}:

$$K_{\text{Manual}} \approx \begin{bmatrix} 902.61 & 0.0 & 703.02 \\ 0.0 & 904.98 & 366.97 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{5a}$$

$$K_{\text{Factory}} = \begin{bmatrix} 918.36 & 0.0 & 661.19 \\ 0.0 & 919.15 & 356.60 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{5b}$$

To assess the difference between the average intrinsic matrix and the factory calibration, we calculated the mean squared error (MSE), which is approximately 10.21. This relatively small MSE indicates that our calibration results are reasonably consistent with the factory calibration. The source of this error could be manifold: imperfections in the checkerboard (not perfectly flat for instance), variations in camera pose during calibration, noise in the captured images, etc. However, since the factory calibrations are often performed under controlled conditions those imperfections could have been avoided, which is probably why the provided factory calibration is probably slightly more accurate.

2) Extrinsic Calibration: Here are the matrices from the three different methods, H_1 is the naive version, H_2 using the accelerometer and H_3 is the auto-calibration using AprilTags:

$$H_1 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & -1 & 0.0 & 240.0 \\ 0.0 & 0 & -1 & 1030.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{6a}$$

$$H_2 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & -0.99 & 0.1225 & 240.0 \\ 0.0 & -0.1225 & -0.99 & 1030.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{6b}$$

$$H_3 = \begin{bmatrix} 0.999 & -0.010 & -0.014 & 3.579 \\ -0.008 & -0.994 & 0.112 & 216.215 \\ -0.015 & -0.112 & -0.994 & 1020.304 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix} \tag{6c}$$

As one can imagine, the manually obtained external matrices, have relatively higher error compared to the AprilTag-based method. This is because manual measurements can be imprecise, influenced by human error, while AprilTags provide a more accurate reference especially in our specific context where we leveraged the redundancy of 35 different points for the AprilTag method.

3) Workspace Reconstruction: To evaluate the accuracy of the workspace reconstruction we used some OpenCV functions to draw the center and the contours of the AprilTags on the bird's-eye view, as well as the projected points of the grid and compare them to their position on the real-time RGB Video frame.

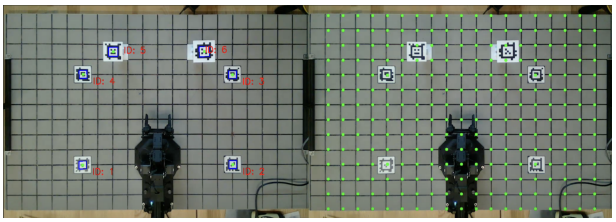

Fig. 13. Visualion of AprilTags contours and projected gridpoints

4) Block Detection: To assess the precision of our block detection system and substantiate its performance, we used a heatmap visualization. This approach involved meticulously positioning blocks at predetermined locations on the grid, thereby establishing a ground truth. Our block detection mechanism then predicted the coordinates of each block's center from its position on the pixel image, leveraging both extrinsic and intrinsic calibration. The objective was to compare the detected coordinates for each block against the ground truth, which represents the precise center position of the block on the grid. This comparison allowed us to gauge the accuracy of our block detection system using the l2-norm of the distance between the predictions and the groundtruth. The configuration used for this figure is the one presented in Figure 2.

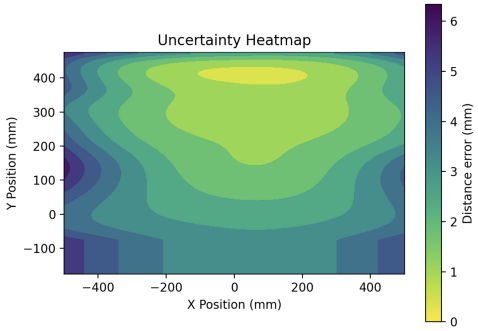

Fig. 14. Uncertainty map to assess the accuracy of the block detector

B. Robot Controls

1) Forward kinematic solution: Given the frames, we were able to determine the screw axes in the base frame. First, we found the M matrix based on the situation where the angle of all of the joints in the arm are 0.

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 424.15 \\ 1 & 0 & 0 & 303.91 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{7}$$

Once the M matrix was found, we then found all of the screw axes for each of the joints as seen in the table below.

joint	ω	v
1	(0,0,1)	(0,0,0)
2	(-1,0,0)	(0,0,103.91)
3	(-1,0,0)	(0,50,200)
4	(-1,0,0)	(0,200,0)
5	(0,-1,0)	(0,65,0)

2) Inverse kinematic solution: For each of the orientations that were defined in figure 11, we derived the formulas to determine the angles of first three joint, denoted by θ_1, θ_2 and θ_3. These included finding various other angles that are universal across the various orientations that are related to the joint angles as denoted below. You can also see what angles they represent in Fig. 15.

$$h = \sqrt{x_w^2 + y_w^2 + (z_w - 103.91)^2} \tag{8}$$

$$k = \arctan(\frac{z_w}{\sqrt{x_w^2 + y_w^2}}) \tag{9}$$

$$\beta = \arccos(\frac{-h^2 + 205.73^2 + 200^2}{2 * 205.73 * 200}) \tag{10}$$

$$\alpha = \arccos(\frac{-200^2 + 205.73^2 + h^2}{2 * 205.73 * h}) \tag{11}$$

where x_w, y_w, and z_w are the coordinates of the wrist. The various numbers are the measurements of the linkages in the arm in mm.

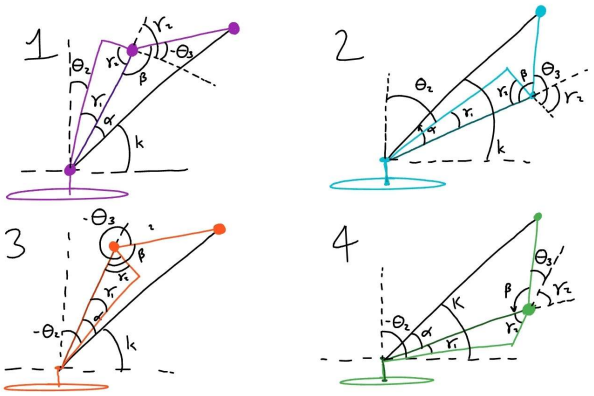

Fig. 15. Stick diagrams of each possible position and the angles needed to find θ_2 and θ_3.

Theta	Pos1	Pos2
θ_1	$-\arctan(\frac{x_w}{y_w})$	$-\arctan(\frac{x_w}{y_w})$
θ_2	$\frac{\pi}{2} - \gamma_1 - \alpha - k$	$\frac{\pi}{2} - k + \alpha - \gamma_1$
θ_3	$-((\gamma_2 + \beta) - \pi)$	$\pi - (\beta - \gamma_2)$
θ_4	$\frac{\pi}{2} - \theta_2 - \theta_3$	$\frac{\pi}{2} - \theta_2 - \theta_3$
θ_5	θ_1	θ_1

Theta	Pos3	Pos4
θ_1	$-\arctan(\frac{x_w}{y_w})+\pi$	$-\arctan(\frac{x_w}{y_w})+\pi$
θ_2	$-(\frac{\pi}{2}-k-\alpha+\gamma_1)$	$-(\frac{\pi}{2}-k-\alpha+\gamma_1)$
θ_3	$-(2\pi-\beta+\gamma_2)$	$-(2\pi-\beta+\gamma_2)$
θ_4	$-\frac{3\pi}{2}-\theta_2-\theta_3$	$-\frac{3\pi}{2}-\theta_2-\theta_3$
θ_5	$-\theta_1$	$-\theta_1$

Although there are four solutions to θ_1, θ_2 and θ_3, we only need one from them. So, in our program, we choose Pos1 as our solution of Inverse Kinematics. θ_1 is computed by $-\arctan(\frac{x_w}{y_w})$ (kinematics.py line 260); θ_2 is computed by $\frac{\pi}{2}-\gamma_1-\alpha-k$ (kinematics.py line 276,280); θ_3 is computed by $-((\gamma_2+\beta)-\pi)$ (kinematics.py line 275,278); here, let $\phi=\frac{\pi}{2}$, θ_4 is computed by $\phi-\theta_2-\theta_3$ (kinematics.py line 282); here, let $\theta=0$, θ_5 is computed by $\theta_1+\theta$ (kinematics.py line 281).

C. Path planning

With the help of block detector, we are able to input the obstacles when we are planning the trajectory. In most cases, the planner will successfully return a near-optimal trajectory. In some cases, the planner will throw out a lengthy trajectory. In extreme cases where the goal is too close to the obstacle, the planner can't find a solution within 1000 iterations. An example with visualizations is shown in Fig. 16.

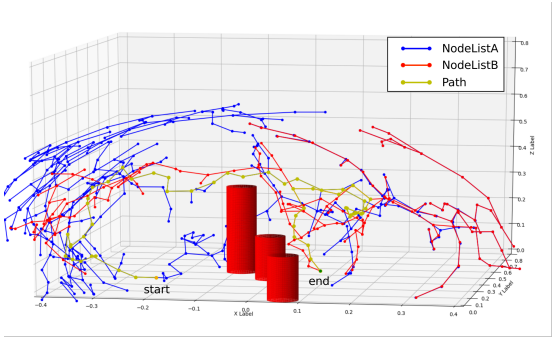

Fig. 16. Motion Planning result from RRT*-connect algorithm.

All the blue lines form the tree growing from the starting point and all the red lines represent the other tree. The orange point is the starting point and the green point is the ending point. The red cylinder is the obstacles in the work space.

IV. Conclusion

Over the course of the first half of the project, we gained familiarity with the arm robot, determined basic parameters for the controls and camera. The second half, we used forward and inverse kinematics as well as some motion planning algorithms for the motion control. And we calibrated the camera and used it and the depth sensor to detect the location, orientation, and color of cubes. By the end of the project, we had fairly accurate autonomous movement of the robot.

A. Competition Functionality

The competition consisted of 4 events and a bonus event: For Pick 'n sort, we completed level 2 successfully, so the arm was able to move the blocks from the positive y half-plane to the negative y half-plane, placing the small blocks on the left and the big blocks on the right. For Pick 'n stack, we completed level 2 successfully. Using our techniques for picking and placing, we can stack the blocks precisely, even for the small blocks. For Line 'em up, where the arm had to line up the blocks in rainbow order with particular constraints. The total length of the line is 300mm, so that a interval of only 10mm is allowed when placing the blocks. So, We set the pose of the gripper to a specific configuration to save the time from adjusting their location afterwards. We also did a clean-up of the workspace before we begin to line the blocks and thus we can make sure there is enough block-free area to make the line. For Stack 'em high, where the arm had to stack a set number of blocks. The result is similar to task2 and we succeeded in finishing the task on level 1 every time.

B. Future Improvement

For the hardware, there were issues with the hardware of the robot which caused some inconsistencies with our performance. Besides the general imprecision of the robot due to the way it was built, there was an issue with one of the motors. The left motor failed after running for a relatively long time. The maximum torque it can provide is reduced.

For the motion planning part, changing the target of the planning algorithm from all the 5 joints to only the first 4 joints of the robot arm may help improve the performance. This is because the last joint only determines the angle of the jaw in local z axis, which does not have much effect on the position of the robot links. What's more, large difference in the last joint angle results in a large norm of the difference between two joint vectors, while their corresponding Euclidean space positions only have small difference, which result in low efficiency of finding nearby joint vectors.

For task 3, level 3, we were asked to manipulate and stack semi-circle and arch blocks. If we had time to apply the grabbing technique, we can center the semi-circle block and grab it accurately. Same with the arch blocks since the accuracy of the stack is important.

REFERENCES

[1] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Wiley, 2005. [Online]. Available: https://books.google.com/books?id=wGapQAAACAAJ