ROB 550 Group 3 Report

Nitish Sanghi, Oscar de Lima
{nitishs, oidelima}@umich.edu

***Abstract*—The ArmLab is a team based project were students implement a robot arm to grab and place blocks. The purpose of the lab is for students to learn basic robotics skills such as how to implement a vision system, forward kinematics, inverse kinematics, trajectory planing and other relevant tasks.**

I. Introduction

For the ArmLab teams of 2 to 3 people work on a robot arm for approximately one month in preparation for a final competition. In order to do these tasks, each team needs to build a vision system to locate the blocks on a flat surface, design a gripper to grasp the blocks and program a robot arm to place the blocks where needed. Before the competition, each team has to complete 4 tasks and show them to the teaching assistants for verification. The first task is to show a working gripper and discuss the design with a TA. The second is to demonstrate that the robot can follow a set of waypoints that are manually recorded by the user. The third task is to demonstrate that the camera calibration is able to locate blocks in the world. The last task is to demonstrate that user can pick a block by clicking on it in the GUI camera display. All four tasks were completed in time by the deadline.

II. Methodology

A. Clamping

Before starting to work on the different functionalities for the arm the team had to ensure that it was safe to operate the robot arm without damaging it. Each joint angle is limited because the arm could hit itself or because of inherent motor angle limitations. In order to determine the range of values each joint could take before hitting the arm:

1) The arm was placed in the initial zero configuration with all links in a straight line upwards.
2) Manual mode was activated and the torque was set to zero.
3) A joint was rotated until it almost reached both upper and lower angle limits. The angles were obtained with the get_positions function (`rexarm.py file, line 127`).
4) The joint was returned to the zero degree position.
5) Step 3 and 4 were repeated for the remaining joints.

By following this process, the joint angle limits obtained were:

Joint	Lower bound	Higher bound
1 (Base)	No limit	No limit
2 (Shoulder)	-1.74 rad	1.74 rad
3 (Elbow)	-1.58 rad	1.58 rad
4 (Wrist 1)	-2.6 rad	2.6 rad
5 (Wrist 2)	-1.7 rad	1.7 rad
6 (Wrist 3)	-5.23 rad	5.23 rad

TABLE I: Joint angle limits for the clamping function

Using the angle limits shown in Table I the clamping function was implemented (`rexarm.py, line 167`). The function takes a list of joint angles as arguments and returns the same list of angles which each value clipped by an upper or lower bound. This function was then called inside the set_positions function (`rexarm.py, line 79`). In this way, every time set_positions was called each angle was clipped to a safe value.

B. Waypoint Follower

Once the clamping function was written and it was safe to send commands to the joint positions, the next step was to test the set_positions function. The idea was to add a new state "execute" to the state machine that would call the function "execute" (`state_machine.py, line 488`). The `execute` function took a list of waypoints as an argument, called `set_positions` on each waypoint and then called pause (`rexarm.py, line 159`) with enough seconds to give the arm time to reach each waypoint. In order to test if the function worked correctly the waypoints used where well known angle positions. For example: elbow at pi/2 rad, shoulder at -pi/4 rad, base at pi rad, etc. A button with the name "EXECUTE" was then setup in the GUI so that the user could start the waypoint follower whenever he or she wanted.

C. Teach and Repeat

The idea of teach and repeat was that the user would set the torque to zero and move the arm to different

positions. After reaching each position the user would press a button in the GUI titled "RECORD" that would retrieve the current position of the joints and append them to a list. The current state of the state machine and the list of waypoints to be followed would be displayed on the bottom of the GUI. Later, the user would be able to press "EXECUTE" and the arm would follow the waypoints recorded in the list using the functionality of the waypoint follower discussed in the previous section. The user could then use the "Teach and Repeat" functionality to play the game of Operation that was provided by the instructor.

D. Trajectory Planning

With teach and repeat implemented, the robot had all the necessary tools to move anywhere the user wanted to. The problem was that this movement was jerky and could have lead to the arm dropping the blocks unexpectedly. In order to avoid throwing the blocks across the room the arm would have to move in smoothly without sudden changes in velocity or acceleration.

A quintic polynomial trajectory was implemented according to the class' textbook instructions [1]. With this approach, the initial position q_0 , the final position q_f , the initial velocity v_0, the final velocity v_f, the initial acceleration a_0 and the final acceleration a_f are specified. Since there are 6 constraints to satisfy, a polynomial with 6 independent variables is necessary.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \quad (1)$$

The velocity is given by

$$v(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \quad (2)$$

The acceleration is given by

$$a(t) = 2a_2 t + 6a_3 t + 12a_4 t^2 + 20a_5 t^4 \quad (3)$$

Because we have 6 constraints, we get the following 6 equations

$$q_0 = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 + a_4 t_0^4 + a_5 t_0^5 \quad (4)$$
$$v_0 = a_1 + 2a_2 t_0 + 3a_3 t_0^2 + 4a_4 t_0^3 + 5a_5 t_0^4 \quad (5)$$
$$a_0 = 2a_2 t_0 + 6a_3 t_0 + 12a_4 t_0^2 + 20a_5 t_0^3 \quad (6)$$
$$q_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 + a_4 t_f^4 + a_5 t_f^5 \quad (7)$$
$$v_f = a_1 + 2a_2 t_f + 3a_3 t_f^2 + 4a_4 t_f^3 + 5a_5 t_f^4 \quad (8)$$
$$a_f = 2a_2 t_f + 6a_3 t_f + 12a_4 t_f^2 + 20a_5 t_f^3 \quad (9)$$

These 6 equations can be put in matrix form like

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 & t_0^4 & t_0^5 \\ 0 & 1 & 2t_0 & 3t_0^2 & 4t_0^3 & 5t_0^4 \\ 0 & 0 & 2 & 6t_0 & 12t_2 & 20t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ a_0 \\ q_f \\ v_f \\ a_f \end{bmatrix} \quad (10)$$

The function `generate_quintic_spline` (`trajectory_planning.py, line 38`) takes as arguments a list with the starting position of the joint angles, a list with the joint angles for the final position and the amount of time T the user wants the trajectory to take. It returns a spline for the position and velocity of the trajectory. The function then sets the value of q_0 to the starting position, q_f to the final position, v_0 to 0, v_f to 0, a_0 to 0 and a_f to 0 for every joint independently. The choice of setting v_0, v_f, a_0 and a_f to 0 comes from the simplicity of not having to worry about whether the initial and final position passed to the function are part of a larger trajectory.The function solves equation 10 to get the values of $a_0, \ldots, a_5$.The function then generates a linear space for the time $t_{span} = [0, \Delta t, 2\Delta t, \ldots, T]$ where $\Delta T = 0.11$ secs. Finally, the function uses equation 1 and 2 to get $q = [q_0, \ldots, q_T]$ and $v = [v_0, \ldots, v_T]$.

In order to execute the position and velocity spline generated by the `generate_quintic_spline` function, an `execute_plan` function (`trajectory_plan.py, line 64`) was implemented. This function takes q and v. A loop is then run over each value of v and the `set_speeds` function (`rexarm.py, line 99`) is called with each value of v. The problem with looping though each position of q in a way that the arm position is only a step away from the value of q being evaluated is that the inertia of the arm and the the inherent properties of the motors make it impossible for the position to change when Δq is too small. The solution is to set the position of the arm angles to the values of q that are a number of steps in the future. This number is passed to the function as the argument `look_ahead`. If the loop over q is so close to the end of the trajectory that there are less steps remaining than the look ahead number, the position of the arm is set to q_f. The default look ahead number used in the project was 15 steps. After calling the `set_positions` and `set_speeds` functions, a pause of ΔT seconds is called.

E. Gripper Design

For the competition, a gripper had to be designed to grab and drop the blocks in different positions. The criteria used by the team to design the gripper was:

1) The gripper is light enough that the motors in the kinematic chain won't be damaged by excessive weight.
2) The gripping surface is flat to maximize the amount of contact we the sides of the blocks.
3) The gripping surface is build from a material with a high friction coefficient in order to minimize the amount of torque the motors have to apply to grab the blocks.
4) The gripper is able to grab and release blocks from above.
5) The gripper is able to grasp and release blocks from the sides.
6) The design of the gripper is modular in a way that it can be disassembled to fix or replace any parts.
7) The gripper won't break or permanently deform under the stress induced from grabbing the blocks.
8) There is free access for at least one connector from the motor to connect to the arm chain.

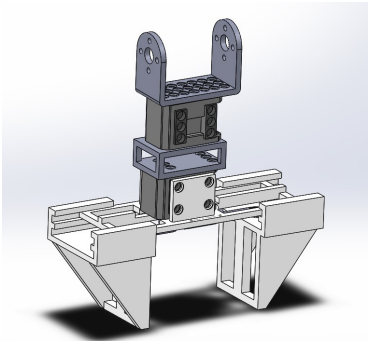

Fig. 1: Gripper assembly design on Solidworks

It order to make the gripper as light as possible holes were placed in non-critical spots of most of the parts. Rubber bands were placed around the fingers to increase their friction and lift the blocks more easily. The grabbing surface for each finger has an area of 40 x 40 mm^2. These dimensions where chosen as a middle ground between having a large surface with many points of contact vs. and having a small surface that wouldn't get in the way of grabbing blocks. All of the parts of the design can be put together using the OLLO rivets that the instructor provided. In order to make the design more complaint to stress a triangular shaped support structure was added to the fingers on each side. To mount the gripper design to the rest of the arm the part given by the TA's in the form of a tutorial was used. The gripping mechanism works by rotating the bottom motor. This motor is connected to a rotating disk. Two links are connected on one end to the disk and in the other to each finger. When the motor turns the disk, the links are forced to move towards the center of the gripper and the fingers follow. In this way, the opening and closing

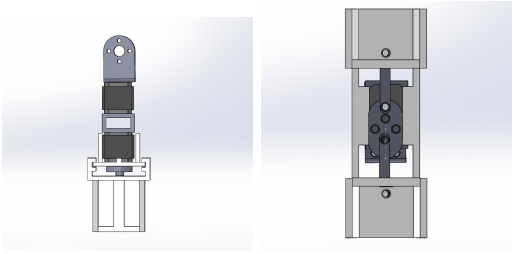

Fig. 2: Gripper assembly. Side (left) and bottom (right) view.

motion is controlled. Instead of using the two allowed motors to grab the blocks directly, the second motor was used to give the arm 6 degrees of freedom. It works by rotating a rigid base that rotates the whole assemble.

F. Forward Kinematics

Forward kinematics is the process through which the position of the end effector is obtained from the values of the joint parameters.

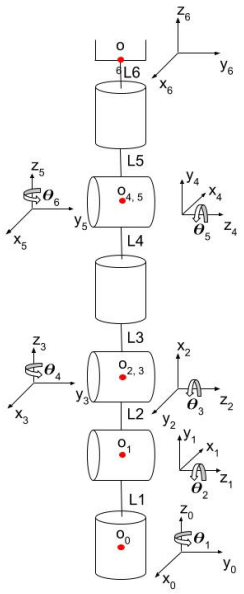

Fig. 3: Schematic of robot arm. Frames 1 through 6 have origins at $o_1, \ldots, o_6$. The link lengths of the arm are $L1, \ldots, L6$. The angle of rotation $\theta_1, \ldots, \theta_6$ are defined by the curved arrows in the positive direction.

In this project, the Denavit–Hartenberg convention was used to set the reference frames for the joints and the end effector. Having chosen the frames as shown in Figure 3, the DH parameters obtained were

Link	a_i	α_i	d_i	θ_i
1	0	$\pi/2$	L_1	$\pi+\theta_1$
2	L_2	0	0	$\pi/2+\theta_2$
3	0	$\pi/2$	0	$\pi/2+\theta_3$
4	0	$\pi/2$	L_3 + L_4	$\pi\theta_4$
5	0	$\pi/2$	0	$\pi+\theta_5$
6	0	0	L_5 + L_6	θ_6

TABLE II: DH parameters table.

The homogeneous matrix A_i is defined as the transformation from frame i to frame $i-1$. The matrix H is defined as the transformation between the end effector and the origin. Multiple transformations $A_i, \ldots, A_1$ can be multiplied to get the transformation from the end effector to the ground frame.

$$H = A_1 A_2 A_3 A_4 A_5 A_6 \tag{11}$$

with c(θ_i) defined cos(θ_i) and s(θ_i) meaning sin(θ_i)

$$A_i = \begin{bmatrix} c(\theta_i) & -s(\theta_i)c(\alpha_i) & s(\theta_i)s(\alpha_i) & a_i c(\theta_i) \\ s(\theta_i) & c(\theta_i)c(\alpha_i) & -c(\theta_i)s(\alpha_i) & a_i s(\theta_i) \\ 0 & s(\alpha_i) & c(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

The `FK_dh` function (`kinematics.py, line 23`) takes 6 joint angles and uses equation 12 and 13 to calculate $A_1, \ldots, A_6$. It also takes in a link number in the case that the user doesn't want to get T_0^6 but would rather get T_0^{link}.

G. Inverse Kinematics

Inverse Kinematics is the process through which we recover the values for the joint parameters from the position and the orientation of the end effector. The difficulty with this process is that, contrary to forward kinematics, there can be multiple joint parameter solutions for one location. The first step is to define a ground frame in which the x, y and z coordinates will be given. In this project Frame 0 from Figure 3 was used as the ground frame. The approach used was to decompose the problem into two solutions. Getting the inverse position of the center of the wrist o_c and getting the inverse orientation of the wrist center. In order for the arm to reach a specific location, x_0 has to be pointing towards the x and y coordinates of the desired position or π rad away from it.

$$\theta_1 = tan^{-1}(y/x) \tag{13}$$

θ_1 is always chosen as $atan2(y_c, x_c)$ and not $\pi + atan2(y_c, x_c)$. In order to solve for θ_2 and θ_3 it is necessary first to compute the position for the center of the wrist. The center of the wrist lies in the intersection of the z axis of the last three joints. In Figure 3, that location falls on $o_{4,5}$. Because the orientation is given we can compute o_c with the end effector position and the last column of the rotation matrix in the following way according to the textbook's instructions [1].

$$o_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} o_x - (L_5 + L_6) r_{13} \\ o_y - (L_5 + L_6) r_{23} \\ o_z - (L_5 + L_6) r_{33} \end{bmatrix} \tag{14}$$

Where L_5 and L_6 are last 2 link lengths and r_{ij} is the element in row i and column j from the rotation matrix. With the coordinates for o_c, θ_3 and θ_2 can be obtained. Using the textbook equations [1] and transposing to our definitions of each angle we get

$$\begin{aligned} \theta_3 = cos^{-1}((o_{c,x}^2 + o_{c,y}^2 + (o_{c,z} - L_1)^2 - \\ (L_3 + L_4)^2 - L_2^2)/(2(L_3 + L_4)L_2)) \end{aligned} \tag{15}$$

$$\begin{aligned} \theta_2 = \pi/2 - tan^{-1}(\frac{o_{c,z} - L_1}{\sqrt{o_{c,x}^2 + o_{c,y}^2}}) - \\ tan^{-1}(\frac{(L_3 + L_4) sin(-\theta_3)}{L_2 + (L_3 + L_4) cos(-\theta_3)}) \end{aligned} \tag{16}$$

The elbow up solutions is always chosen because it interferes less with the blocks. θ_3 is always positive in this project.

To get angles θ_4, θ_5 and θ_6 in is necessary first to get the rotation matrix R_6^3. Since the matrix R_6^0 is already known and R_3^0 can be obtained with forward kinematics, R_6^3 is solvable

$$R_6^0 = R_3^0 R_6^3 \implies R_6^3 = (R_3^0)^{-1} R \tag{17}$$

With R_6^3 we can solve for θ_5 with the standard form of a rotation matrix given in the textbook [1].

$$\theta_5 = tan^{-1}(\frac{\pm\sqrt{1 - r_{3,3}^2}}{r_{3,3}^2}) \tag{18}$$

Where $r_{i,j}$ corresponds to R_6^3.

The positive square root inside θ_5 is always used. For the two remaining angles we have to consider three cases. The first case is if $r_{13} = r_{23} = 0$. In this case the desired position is in a singularity and there are infinite solutions for θ_4 and θ_6. The second case is when $sin(\theta_4) > 0$. In this case

$$\theta_4 = tan^{-1}(\frac{r_{23}}{r_{13}}), \theta_6 = tan^{-1}(\frac{r_{32}}{-r_{31}}) \tag{19}$$

The last case is when $sin(\theta_4) < 0$. In this case

$$\theta_4 = tan^{-1}(\frac{-r_{23}}{-r_{13}}), \theta_6 = tan^{-1}(\frac{-r_{32}}{-r_{31}}) \tag{20}$$

The function `IK` (`kinematics.py, line 138`) takes the desired final position and a rotation matrix. It

returns $\theta_1, \ldots, \theta_6$ using equations 13, 14, 15, 16, 17, 18, 19 and 20.

H. Camera Calibration

To manipulate objects the robot arm requires a perception system which would take in raw sensor inputs and output data which the robot system will use to localize itself, the objecs, and the workspace in general. For this project a Kinect 2.0 camera and depth sensor was used to determine spatial location of objects in the workspace. Kinect 2.0 outputs depth and 2D scene information which can be processed to generate usable inputs for the robot system. But before the outputs of Kinect 2.0 can be used, the camera and depth sensor need to be calibrated. Camera calibration is broken down into two phases: a) Camera Image and Depth Image alignment and b) Camera Image and World Space alignment. Both of these phases put together locate an object in 2D space and the height of the object, effectively working as a 2.5 D vision system. Camera calibration was implemented as a state of the `statemachine` in "statemachine.py" which calls the calibrate(self) function to begin calibration. To calibrate the camera system, the user clicks on the "CALIBRATE" button in the GUI and the system goes into the calibration state. When prompted by the GUI, the user clicks on 5 known pixel points in the RGB frame and the Depth frame of the GUI. These points are used to align the Depth and RGB frames and also the image and workspace frames which is discussed in the next few subsections.

1) Depth Image and Camera Image Alignment: The depth sensor and RGB camera sensors are located close but are offset in the same plane. The pixel coordinates in the depth and vision frame need to to aligned to extract useful information about the workspace. If the camera system is directly above the workspace at a fixed distance from all object, as was the case, an affine transformation $(equationsbelow)$ can be used to relate depth frame pixel coordinates x and y to camera frame pixel coordinates u and v.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{21}$$

To find the transformation matrix, known pixel points in the depth and vision space are used to generate a system linear equations which is then solved to get the elements for the transformation matrix. Elements of Matrix A are vision frame coordinates and of vector b are depth frame coordinates of the area of the workspace area. Vector X contains the elements of the affine transform from the matrix in $(equationabove)$.

$$AX = bX = A{-}1bX = (ATA){-}1ATb \tag{22}$$

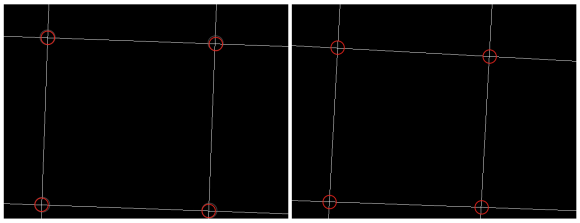

Fig. 4: The intersections are corner points in the depth and rgb frame which need to align. Depth Frame (left) and RGB Frame (right).

The function getAffineTransform(self, coord1, coord2) is called in the calibration state to find the affine transformation. The workspace corner pixel points previously collected in the RGB and Depth frames are input into the getAffineTransform function.

The function then creates matrix A and vector b using the pixel coordinates to generate a system of linear equations which is solved using a linear algebraic equation solver to give the elements of the transformation matrix. The function then arranges the elements to form the transformation matrix and stores it in a class variable to be used to transform the depth frames.

2) Camera Image and Workspace Calibration: Localizing objects in the workspace requires a transformation which can relate the RGB frame pixel coordinates and the workspace scene coordinates. To find the transformation matrix, the optical and image sensor properties need to be captured. A script provided as a resource `camera_cal.py` was used to generate the camera intrinsic matrix and also the distortion coefficients. The kinect is positioned so that the z axis of the camera frame aligned center z axis perpendicular to the workspace. The `affineworspace` function uses the RGB frame and workspace points to generate a transformation matrix to transform camera and workspace coordinates.

3) Calibration Accuracy: Depth and RGB Frame Transformation: Accuracy of the calibration between the Depth and RGB Frame was evaluated by using blocks of fixed dimensions stacked at different heights at different locations. It was noted that the height of the flat workspace area varied at different points. The variation in the workspace area height has to be accounted for the block stack heights.

Table III lists example data for actual heights, stack heights, Local Area Height, and the Net Error for stacks illustrated in Figure X. Table III presents the absolute average and standard deviation for 1 and 2 block stacks as illustrated in Figure Xx.

4) Calibration Accuracy: Workspace and RGB Frame Transformation: The accuracy of the calibration between the workspace and RGB Frame was done using known

Blocks	Height	Height measured	Error
1	38.2	35	-3.2
2	76.5	76	-.5
3	114.8	115	.2
4	153	157	4

TABLE III: Height examples. Measurements in mm

Blocks	Average Error	Standard Deviation
1	1.29	1.84
2	1.63	1.73

TABLE IV: Absolute error and standard deviation in mm.

fiducials. Here a graduated inch pad has been used. The pad is placed at different locations and pixel coordinates are capture for different points on the pad. The distance values are calculated using the transformed coordinates which the system estimates for the points and the distance values are compared. The absolute average error was found to be approximately 5 mm.

5) Block Detection: The block detection algorithm has 3 steps. Step 1 is isolation of the blocks in the workspace. Step 2 is identification of the block colors and step 3 is converting the block information into usable information for the robot. The functions for step 1 and 2 are continuously running in the background. Step 2 uses a dictionary as the data structure to aggregate and store block color and centroid position in pixels. The dictionary is called by step 3 when a task is assigned to located the block in the pipeline to accomplish the task. In step 3 the pixel coordinates are used to find the height of the block and transformed into workspace coordinates for trajectory planning.

Step 1: Block Isolation To begin with, isolating the blocks in workspace using RGB frame is difficult due to the amount of pixel data present and to do so reliably

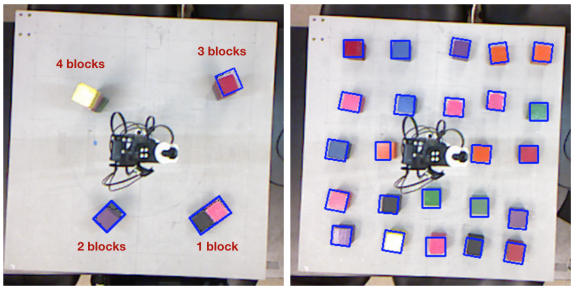

Fig. 5: Example of accuracy check for depth RGB transformation (left) and 1 Block stacks for accuracy check using multiple locations (right).

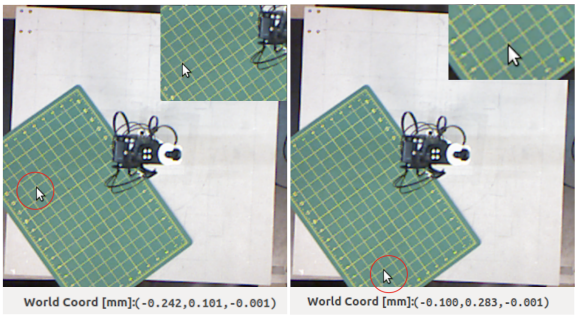

Fig. 6: The workspace coordinates found using the mouse pointer were used to compute the distance between various point on the pad as illustrated

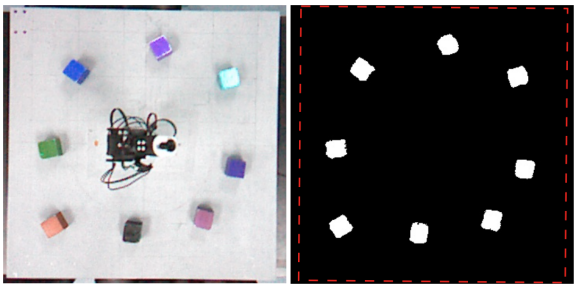

Fig. 7: RGB Frame image (left) and blocks masked image (right).

will require sophisticated algorithms. In order to avoid doing so, information from the depth frame, which has a single channel, is used to isolate the blocks. Since the height of a block is known, a workspace volume stretching from half the height of a block to a certain number of block height is isolated using the depth frame. Depth frame measured values which do not fall in this volume are set to zero. The remaining elements in the depth frame have finite non-zero values. Function convertBlockDepthFrame(self) implements this filter and creates a block mask with blocks isolated.

Step 2: Localization and Color Bucketing Multiple functions are called to localize and color bucket the blocks. The primary function called is detectBlocksInDepthImage(self) where the block color is identified using the colorbuckets(self) sub-function, the orientation and the block centroid pixels using the blockPose(self, color, block) sub-function. To identify the colors, a RGB frame is captured, Gaussian blurred to handle image noise, and converted into HSV image format. The Hue channel of the converted image is used to identify red, green, blue, yellow, pink, purple, and orange blocks. Pre-determined hue color space values are used to generate 7 color specific hue images. The block mask generated using the depth frame filter is

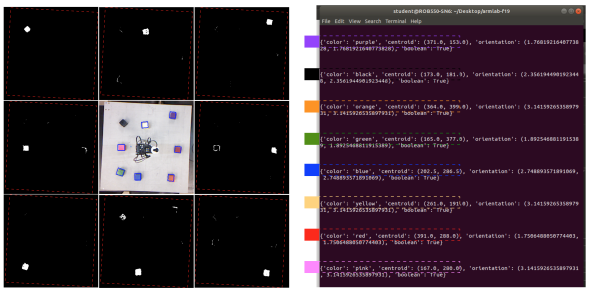

Fig. 8: The center RGB frame image is the location of the blocks. The 4 binary images left to right top row with pixel coordinates in parentheses are black (173,181), yellow (261,191), purple (371,153), and red (391,288). The 4 binary images left to right bottom row are pink (167,280), green (165,377), blue (202,286), orange (364,399). The block colors are identified in the running list printed in the terminal. The blocks are located in the pixel locations are located in the pixel locations specified with pixel (0,0) located in the left top corner.

used to isolate the blocks in each of these images. The value channel of the HSV image is used to isolate the black block, which is then masked as the other 7 derivative images. The color bucket function returns 8 binary images which are then used to localize the blocks. The blockPose function is called on each of the 8 binary images separately. In the function contour of the block is found, which is used by the meanPose(self) to find the centroid and the orientation of the block using two sub-functions centroidBlock(self,contour) and orientationBlock(self,corners). blockPose function stores the values of all the blocks found in a dictionary which can is used for task planning.

Step 3: Coordinate transformation In the final step the block pixel coordinates are converted to world coordinates in by a function in statemachine.py called worldCoordinates(self,x,y) where x,y are pixel coordinates. The function returns 3 values locating the block in XYZ coordinate system centered at the centroid of the flat workspace.

6) Block Detection Accuracy: Determining the correct color for a block is paramount to task completion. Tuning of the hue color range for different colors was done to enable robust block color identification. To test the accuracy blocks with hue colors close to each other like orange and yellow or red and pink were test multiple times at different locations to check color identification. Based on the results the hue color parameters were tuned to get robust identification. Finally, all the blocks were placed together on the board and the location of the blocks was printed in the terminal to check if the correct colors had been identified. The result is captured in figure XX.

I. Motion planning

Planning the motion of the arm for the competition involved using inverse kinematics, camera detection and trajectory planning together. The planning starts with the camera detection which returns the location of the target block in world coordinates. Next the inverse kinematics takes that position as an argument to return a set of angles. The orientation of the hand was set to grab looking downwards when possible. In Figure 3, the axis Z_6 was intended to point towards the ground. If that orientation was out of reach for the arm, the robot would then approach the blocks from the side with Z_6 parallel to the ground pointing from the base of the arm to the block. The arm would then open its gripper and move towards the block. It would then close the gripper and move 10 cm away from the table in the upwards direction. The arm would then move to a point 3 cm above the destination. Finally the block would open the gripper to release the block. Multiple variations of this algorithm were used for the different competitions.

III. Results

A. Testing

1) Path smoothing: In order to test the trajectory planing, the normal movement of the arm was recorded using the set_positions function in order to establish a baseline. The initial position for the test is the zero configuration with 0 for all angles. The final position was set to 1.5 rad, 1 rad, 0.4 rad, 0.7 rad, 1.15 rad and 3.7 rad corresponding to the base, shoulder, elbow, wrist 1, wrist 2 and wrist 3 joints respectively.

2) Kinematics: In order to verify the accuracy of the forward kinematics 15 joint configurations where chosen. For each configuration the accuracy of the forward and backwards kinematics was measured in one test. The arm was positioned at each of these configurations and the location of the end effector was measured with the caliper provided by the instructor (The precision of the caliper is ± 1 mm but it is important to consider that there is a substantial increase in the margin of error because of human imprecision in aligning and observing the calipers accurately). Then, the locations are passed to the inverse kinematics function with an arbitrary orientation to obtain a set of angles that can reach that configuration. The set of angles was passed to the forward kinematics function to obtain a prediction of the location of the end effector.

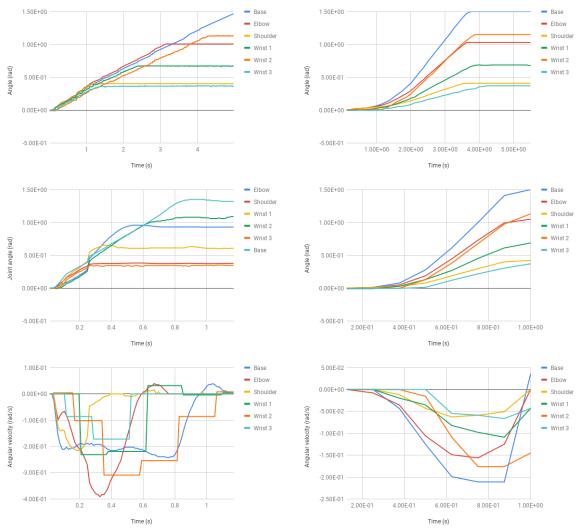

Fig. 9: Trajectory planing tests. The first test was done at a slow speed of 0.5 rad/s (Top left). The same movement was then tested with trajectory planning (Top right). The second test was done at a fast speed of 1.5 rad/s (Middle left). That test was then done with trajectory planning (Middle right). The velocities at high speed for the trajectory planning (bottom left) and no trajectory planning (bottom right) cases were also recorded.

Angle (deg)	X	Y	Z	$X^{'}$	$Y^{'}$	$Z^{'}$
E: 90	197.2	0	137.7	199	0	145
E:-90	-199.0	0	140.0	-199	0	145
E:45	155.7	0	292.2	141	0	286
E:-45	-136.8	0	292.14	-141	0	286
S:90	268.6	0	37.1	300	0	44
S:-90	-292.6	0	35.7	-300	0	44
S:45	210	0	248.2	212	0	256
S:-45	-215.2	0	247.2	-212	0	256
B:90,S:45	0	205.1	234.2	0	212	256
B:90,S:-45	0	-225.1	236.7	0	-212	256
B:90,S:60,E:30	0	267.5	105.2	0	286	94.5
E:90, W2:-90	106.2	0	217.8	113	0	231
E:90,W2:90	124.2	0	63.2	113	0	59
E:90,W1:-45,W2:90	123.2	-76.2	79.2	113	-60.8	84.2

TABLE V: Forward and inverse kinematics test values. E means elbow, S means shoulder, B means base, W1 and W2 refer to the first 2 wrist joints.

The last step was to calculate the error

$$Error = \sqrt{\frac{\sum_{i=1}^{n}((x_i^{'} - x_i)^2 + (y_i^{'} - y_i)^2 + (z_i^{'} - z_i)^2)}{3n}} \quad (23)$$

The error calculated for the table above was 9.53.

3) Gripper: The process for designing and printing the gripper was an iterative where changes would be made, then printed again, then tested, the redesigned and so on. The initial design made for the gripper was very different from the one shown in Figure 1. The first issue that became evident in the gripper was its weight. The design was too large so it got redesigned to a smaller one with many holes to make it lighter. The second issue that became evident was that the part designed to connect the lower motor to the rail was to frail. It kept breaking when removing the support plastic the 3d left on it. It got redesigned with 2 solid squares that go on the sides of the motor and have 4 holes where the OLLO rivets go. The third issue that became apparent was that the space in the finger where the rails where supposed to go was to narrow. It got expanded by a few millimeters. The final issue was that the support structure connecting the two motors was designed in a way that made it impossible to place the OLLO rivets comfortably. Under recommendation of the instructor, one of the parts provided in the lab was used instead.

IV. Discussion

The results for the competition were unsatisfactory. The team tied for the last place with other 2 teams in the section. For the competition it was clear that neither the inverse kinematics nor the camera calibration were working properly.

A. Performance

1) Pick and Stack: The first event attempted was pick and stack. At first the arm kept failing to find the correct location of the blocks in the table. At the time the camera calibration was being done with the help of APRIL tags but that method stopped working before the competition. The causes of this malfunction with the April tags is still unknown. Under heavy time pressure, the team decided to revert back to the old method of calibrating the camera by clicking on the corners of the table and finding the affine matrix. The problem with this method was that it wasn't accurate. The location specified by the camera had a large margin of error and the hand kept missing the targets. The only solution to this problem was placing the blocks close to the arm where the localization worked better. The inverse kinematics wasn't working great either. The inverse kinematic was designed so that the arm grabbed the blocks from above when close and from the side when the blocks where far. The hand worked well most of the time for blocks that were but badly for far away blocks. Our solution was to place all three blocks close to the arm and set the specified stacking position close to the arm too. Even then a slight offset had to be hard coded to successfully stack the blocks.

2) Line em' up: On top of the issues described in the Pick and Stack section, the camera calibration wasn't able to differentiate between colors very often. Because of this the second task proved impossible. Our solution was to try to hard code the initial positions of the blocks we needed to grab but the time constraints didn't allow to do that.

3) Gripper performance: The performance of the gripper in all the competitions was excellent. The gripper could easily grab blocks and take them anywhere and the blocks didn't get stuck in the fingers when attempting to drop them. The decision of using rubber bands to increase the friction of the fingers was a very good one. The gripper also never broke or bent under pressure.

B. Improvements

There are many improvements that could have been made to make the arm work better. The inverse kinematics should have been tested more for locations that were farther away. More testing could have been done to improve the orientation in which the gripper grabs the block. Errors in the choices for the angles in the inverse kinematics had the wrist rotating unnecessarily to reach different locations. Another possible improvement would have been using the orientation of the blocks and aligning the angle of approach of the gripper to that orientation.

With respect to the calibration a more accurate detection coming from use of the April tags could have made a big improvement. Also better color detection would have allowed us to participate in the other events.

With respect to the trajectory planning, all movements between positions in the competitions were set to take 3 seconds. A function that could calculate a more appropriate duration according to the distance between to waypoints would have made the arm faster. This could have resulted in more points in the first competition.

V. Conclusion

To complete the ArmLab every team member had to learn concepts that are central for any roboticist. Some of these concepts were rotation matrices, homogeneous transformations, camera projections transformations, DH parameters, forward kinematics, inverse kinematics, trajectory planning, state machine and serial communication. These concepts were also applied to program a robot to grab and drop blocks. For that teams had to build a vision system to recognize the color and location of the blocks, a gripper to grasp them and multiple motion functions to move the arm around. In the end the team was able to complete the checklist requirements for the lab and the first event in the competition.

References

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