Team 3 ROB 550 Armlab Report

Bo Xue, Di Chen, Haokun He

***Abstract*—Robotics arms are of great importance nowadays in robotics application. Grasping and placing things accurately, these mechanical arms can truly make our life convenient and even do more than human arms. These arms can work under harsh conditions in which human cannot stay. In this lab, we build algorithms for an arm with 6 servos to fulfill different tasks including detecting, grasping and placing blocks. We finish the tasks of camera calibration, forward kinematics, inverse kinematics and block detecting. In our GUI, we make the access to autonomous execution which performs efficient and accurate. Besides, the arm has also some imperfectness when executing tasks and we propose a solution to this.**

Keywords- Forward Kinematics, Inverse Kinematics, Object Detection, Camera Calibration.

I. Introduction

ROBOTICS arms are used in a large variety of fields like underwater mining, circuit building and medical robots. These tasks are often complicated and require various capability including object detection, world frame transformation, and kinematics ability. In this lab, we build algorithms for an arm to accomplish the following different tasks.

- Arm movement teach and repeat
- Forward and inverse kinematics
- Camera calibration and block detection
- Different grasping and moving tasks including lining blocks and stacking them up in a specific color order, sliding a block stack along specific lines

In this report, the first section introduces the application environment and main tasks to fulfill. Section II illustrates basic principles for each task. Section III gives the result and performance evaluation for the tasks. The final section make a discussion about the performance and proposes possible solutions to the potential problems.

II. Methodology

A. Teleoperation & Motion Planning

1) Teleoperation: A graphic user interface (GUI) has been provided to us. By invoking *control_station.py*, we could easily manipulate the arm to execute tasks we set.

2) Path Smoothing: A *set_positions* function is given to us to let the arm move to the desired position. However, the speed will remain constant along the whole way, leading to ungraceful motion especially at the starting and ending position. To smoothly move the arm, we implemented cubic polynomial trajectories on the arm. The total time for the motion is determined by dividing the maximum displacement of the joints by *max_speed*. When the total time is known, we can then calculate the number of waypoints along the trajectory by using $\frac{T_{total}}{dt}$.

Now that we have the number of waypoints, we may need to calculate their positions and corresponding speeds, with the constraints (q_0, q_f, v_0, v_f) = (*initial waypoint, final waypoint, 0, 0*)

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{1}$$

$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2 \tag{2}$$

which can be simplified as:

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix} \tag{3}$$

And we can find the coefficients by

$$a = M^{-1} b \tag{4}$$

When a is known, we can then determine positions q and speeds $\dot{q}$ by using Eq.1 and 2.

B. Grasp Execution

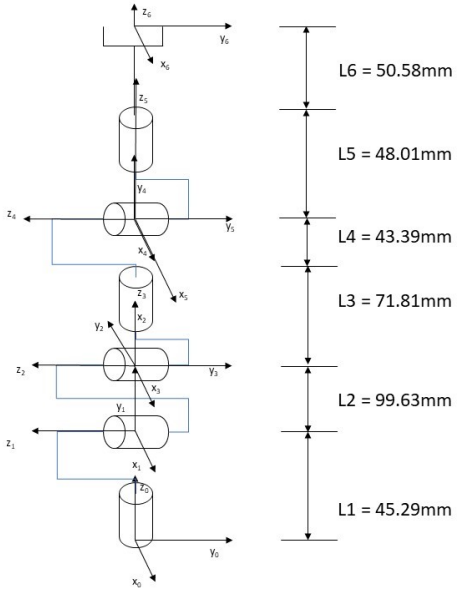

Fig. 1: Configuration of the arm

Joint	θ(rad)	d(mm)	a(mm)	α(rad)
Base	Θ* + π/2	L1 = 45.29	0	π/2
Shoulder	Θ*+ π/2	0	99.63	0
Elbow	Θ*- π/2	0	0	-π/2
Wrist	Θ*	L3 + L4 = 115.2	0	π/2
Wrist2	Θ*	0	0	-π/2
Wrist3	Θ*	L5 + L6 = 98.59	0	0

Fig. 2: dh_table

1) Forward Kinematics: The configuration of the Rexarm robot is shown in Fig. 1. We need to find the workspace coordinates $X = (x, y, z)$ of the endeffector based on joint angle (arm configuration). Here Denavit–Hartenberg contention (DH convention)[1] is applied. Frames of reference are built as shown in Fig1. Four DH parameters θ, d, a and α for each joint are generated to form DH table as shown in Fig. 2.

In this convention, DH parameters are used to generate the homogeneous transformation matrix, and it is given in Eq.1. Then we used these four homogeneous transformation matrix to form the transformation matrix from endeffector coordinate system to global coordinate system shown in Eq.2.

$$A_i = Rot_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} Rot_{x,\alpha_i} \tag{5}$$

$$T^{world}_{endeffector} = A_1 A_2 A_3 A_4 A_5 A_6 \tag{6}$$

In this way, once the DH parameters of all joints are given, we can use the above equation to deduce the endeffector position under world coordinate.

2) Inverse Kinematics: Contrary to forward kinematics, in inverse kinematics, desired endeffector position, saying $X = (x, y, z)$, is given, and corresponding configuration $\theta = (\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$ should be deduced. The deduction is shown as below. First of all, as we only care about the position and the orientation of the endeffector, the arm can be simplified as a 6 DoF arm (gripper excluded). Furthermore, as last three DoF axes intersect at a point o_c, the problem can then be decoupled into two inverse kinematic problems.

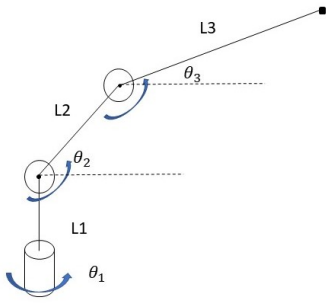

Fig. 3: Schematic of the first three joints

Fig. 3 shows arm with base, shoulder, elbow and wrist. It turns out to be a simple three-link robot problem. As given in Spong book reference, we can first calculate the base joint angle given endeffector world coordinates x_w and y_w:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} o_x \\ o_y \\ o_z \end{bmatrix} - \begin{bmatrix} d_6 r_{13} \\ d_6 r_{23} \\ d_6 r_{33} \end{bmatrix} \tag{7}$$

Where

$$d_6 = L5 + L6 \tag{8}$$

Refer to *kinematics.py* line 147-148

and r_{13}, r_{23}, r_{33} are the entries of the last column of rotation matrix R^0_6.

$$\theta_1 = arctan2(y_c, x_c) - \frac{\pi}{2} \tag{9}$$

Refer to *kinematics.py* line 154-158

Here $\frac{\pi}{2}$ is subtracted because initially the base is along y-axis. Then we further calculate θ_2 and θ_3 using Law of Cosine.

$$\theta_3 = -arccos \frac{\hat{L}^2 - L_2^2 - L_3^2}{2 L_2 L_3} \tag{10}$$

Refer to *kinematics.py* line 159-168

Where

$$\hat{L} = \sqrt{(z_c - L1)^2 + x_c^2 + y_c^2} \tag{11}$$

$$L3 = L3 + L4 \tag{12}$$

Then θ_2 can be deduced given $\theta 3$

$$\begin{aligned} \theta_2 = {} & arctan2(z_c, sqrt(x_c^2 + y_c^2)) \\ & - arctan2(L3 sin(\theta_3, L2 + L3 cos(\theta_3))) \end{aligned} \tag{13}$$

Refer to *kinematics.py* line 170-171

As $(\theta_1, \theta_2, \theta_3)$ are all known, we can deduce the rotation matrix R^0_3 by forward kinematics introduced before. Also, R^0_6 is known as it can be easily deduced from the known endeffector world frame. The rotation matrix and configuration of the next part of the decoupled problem are shown as below:

$$\begin{aligned} R^3_6 &= R^{0\,-1}_3 R^0_6 = R^{0\,T}_3 R^0_6 \\ &= \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \end{aligned} \tag{14}$$

Refer to *kinematics.py* line 179

And we discuss two situations where we have different values for θ_4, θ_5, θ_6: If at least one of the r_{13}, r_{23} is not zero, we have

$$\theta_4 = arctan2(r_{23}, r_{13}) \tag{15}$$

$$\theta_5 = arctan2(-\sqrt{1 - r_{33}^2}, r3) \tag{16}$$

$$\theta_6 = arctan2(r_{21}, r_{11}) \tag{17}$$

Refer to *kinematics.py* line 190-193

else, we have

$$\theta_4 = \theta_5 = 0 \tag{18}$$

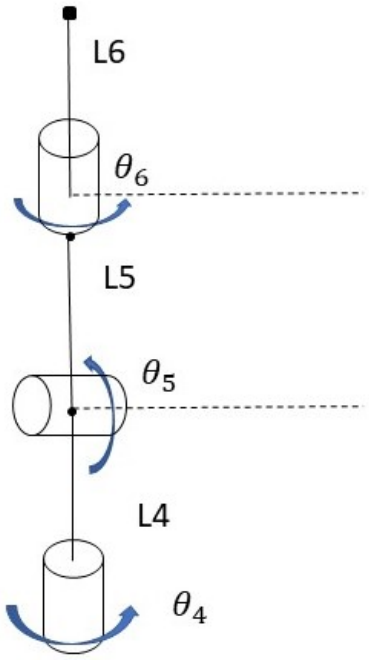

Fig. 4: Schematic of the last three joints

$$\theta_6 = arctan2(r_{21}, r_{11}) \tag{19}$$

Refer to *kinematics.py* line 194-197

Now that we can deduce the exact configuration space given endeffector position and orientation, we need to consider the proper orientation of the block to grab it. As a block is symmetric along every axes, its orientation can be determined in many different ways, as long as it is within the workspace of the arm.

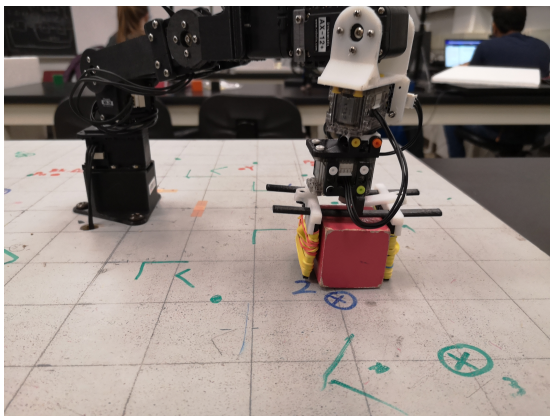

Fig. 5: Pose 1

As shown in Fig. 5, the easiest way and perhaps the most reliable one is to grab the block from its above. In this case, z-axis of the block is heading down. This convention reached almost 100% chance of successfully grabbing a block. However, this method has a huge disadvantage. The workspace is rather limited. When the block is placed 200mm away from the center, the gripper could hardly reach the desired position and chance of failure increased drastically.

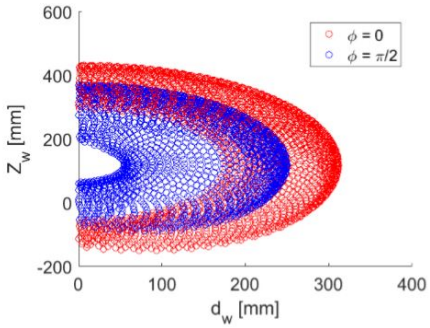

Fig. 6: Pose 1 (blue) and pose 2 (red) workspace

To overcome this problem, another endeffector posture has been introduced. z-axis is set to be parallel to the plane. In this way, the arm can cover points even at (200, 200, 200). In other words, 90% of the space less than 200mm above the board can be covered.

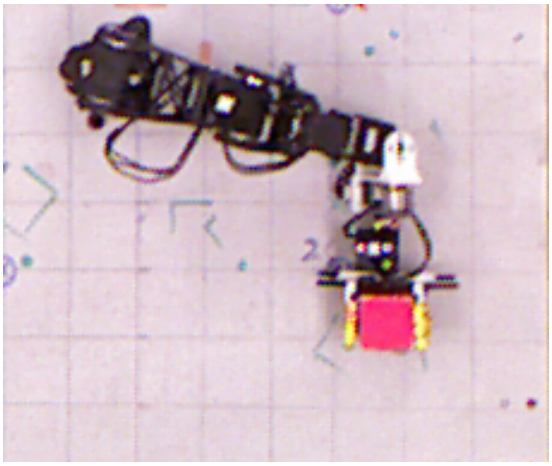

Fig. 7: Pose 2

C. Object Perception

Camera intrinsic matrix obtained from python function:

$$\begin{bmatrix} 533.9231992 & 0 & 311.70734338 \\ 0 & 533.29448952 & 266.38756735 \\ 0 & 0 & 1 \end{bmatrix},$$

and the affine (homogeneous) transformation from depth frame to rgb frame is:

$$\begin{bmatrix} 9.17 \times 10^{-1} & 1.43 \times 10^{-4} & 1.15 \times 10^{1} \\ -4.03 \times 10^{-3} & 9.20 \times 10^{-1} & 4.88 \times 10^{1} \\ 0 & 0 & 1 \end{bmatrix}.$$

To get the depth calibration function, here we do not make any additional assumptions. Instead, we find the 4×4 homogeneous transformation matrix directly from

camera frame to world frame with 8 calibration points. The procedure is detailed as follows:

Depth calibration procedure:

1) Use mouse to click the 8 calibration points with known world frame coordinate and get the image (rgb) frame value.
2) Use the affine transformation matrix from depth frame to rgb frame to register depth frame to rgb frame. After this step, we can get the correct depth value in rgb frame.
3) The depth value from depth frame is of range 0-2047, whereas in the provided d to Z_c transformation function, d is of range 0-1023. Consequently, we first clip the depth value, and then use the provided function:

$$Z_c = 0.1236 * \text{np.tan}(d/2842.5 + 1.1863)$$

to convert d to Z_c.
4) Use the inverse of the camera intrinsic matrix to transform the image (rgb) frame to camera frame.
5) Use the resulting camera frame coordinate and the known world coordinate to get the 4×4 affine transformation from camera frame to world frame.
6) After obtaining the 4×4 affine transformation from camera frame to world frame, the whole calibration procedure is finished, and we can get the world coordinate of any point on the image.

Blob Detection Implementation:

To implement the blob detection, we use python openCV **SimpleBlobDetector** function [2] after converting the 1-layer depth frame into a 3-layer image. The SimpleBlobDetector provides the center of the blob in image frame. With the center of each blob in image frame, we can further determine the color using the rgb and hsv value of the center, draw the contour, determine its orientation, as well as get the corresponding world frame coordinate using the aforementioned transformation.

To limit false positive detection, we use the following procedure:

1) We adjust openCV SimpleBlobDetector parameters. Particularly, we set the minimum and maximum area to be close to the area of the blob to filter anything that is smaller or larger than our threshold.
2) After applying the SimpleBlobDetector, the keypoints (center of each blob) are provided by this function. To reliably draw the contour around the blob and filter other noises, we first crop the image to a square around the center of each keypoint, and then use dilation followed by erosion for Morphological operations.
3) After the Morphological operations, we use the threshold function to find the contour of each blob. However, since it is still possible to have noise in the image after Morphological operations, we only extract the contour that has the maximum items to prevent false detection and increase robustness.
4) Since our focus is on the board, and we do not care about the blobs or other objects outside the board, as the last step, we limit our blob detection range to only be on the board as well as limiting with depth value to prevent detecting our robotic arm.

To reliably detect color, we use the following strategy:

1) Since in HSV, since only the H component contains information about the absolute color [3], we first use h value to differentiate *orange*, *yellow*, *green*, and *blue*, where their h values are all smaller than 130.
2) However, *pink*, *red*, *purple* and *black* are hard to differentiate with h values (their h values are all greater than 130), but the difference in their r values are large. Consequently, we use r value to differentiate these 4 colors.

D. Autonomous Execution For Final Bits and Pieces

- **Pick and Stack**
  In this event, we first use the block detecting function to get all the three blocks' world-frame location and orientation (used for the gripper to grasp in the correct direction). Then, we generate three destination for the three blocks in a stack which locates on the negative direction along x-axis. At last, we place the first detected block to the lowest destination, the second block to the second level, and the third block to the third level.
  For each movement, we set a mediate point between the initial point and the goal point. When moving the block to the destination, the arm first move to the mediate point just 4cm above the destination, then move to the destination. When getting from the last destination to the next initial point, it also first arrives at the mediate point then go to the next initial point. In this way, the arm can avoid hitting other blocks and behaves rationally.
- **Wacky Pick and place**
  In this task, we first place the two frames on the platform and one block on one of the frames. Then we detect the two blocks (because the frame has a black block area, it can also be detected), if the first detected block is the frame with a block on it, set the other block as the destination and the first block as the initial point, vice versa. Finally, move the block from the initial point to the destination with also a mediate point in the track.
- **Line Them Up**
  In this event, the stacks can be at most 3 levels' high. Thus we keep detecting the upper level blocks and moving them to the right position of the destination line, while record the placed blocks' colors in the "used" list, move the block only if the block is not in the "used" list, stop the loop until all the detected blocks' colors are in the "used" list. When moving the blocks, use the mediate points illustrated above to avoid hitting others.
- **Stack Them High**

In this event, the stacks can be also at most 3 levels' high. We first keep moving the upper level blocks to the ground level until all the detected blocks are on the ground level. In this process, the ground level destination is generated from a candidate location list by selecting the points where there are no other blocks in the neighbor(6cm actually). Now all the blocks are on the ground level, we choose the blocks in the required color order to stack them up at an assigned location. In this process, we give each detected block a sequence obeying the required color order, which can make the algorithm finish the task with only one detection.

- **Block Mover**
  In this task, we are required to move a stack of blocks along some line without changing the orientation greatly on the path. In order to make the blocks' orientation almost unchanged, we rotate the arm to adjust it to another pose every 90°. Then we assign the destination for the blocks every 5cm along the path and smooth the trajectory planner.

III. Results

A. Teleoperation & Motion Planning

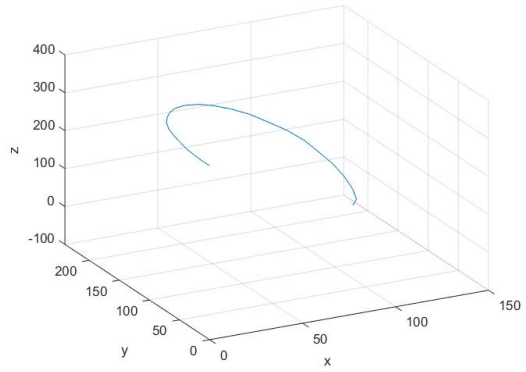

Fig. 8: Endeffector position at high speed with path smoothing

We executed a motion plan from world coordinate (0, 0, 358) to (150, 120, 0). From Fig. 8 - 13, we can see that the motion smoothing part is quite satisfactory. In Fig. 12, there is an outlier. The occurrence of outlier is perhaps due to the position feedback inaccuracy. But overall, with path smoothing, the position curve is quite smooth compared to the one without smoothing. The velocity magnitude, though still a bit sharp, is slightly better than the one without any smoothing.

B. Grasp Execution

To validate the *kinematics* part, the way we did was to randomly select several points on the board (Fig). As the board has accurately grid on it, we could determine the ground truth of the ending point under world frame with the assistance of grid and caliper.

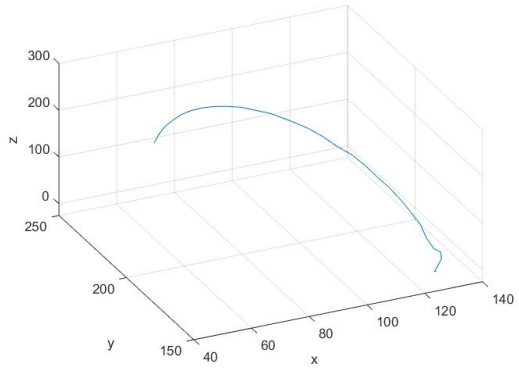

Fig. 9: Endeffector position at low speed with path smoothing

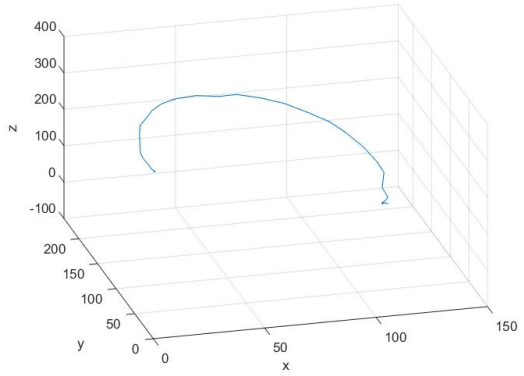

Fig. 10: Endeffector position at high speed without path smoothing

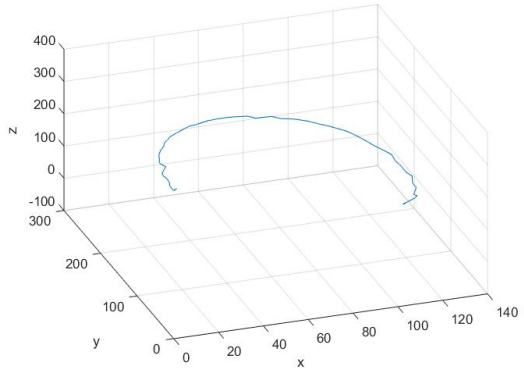

Fig. 11: Endeffector position at low speed without path smoothing

As we can see, errors exists. Reasons can the bad measurement of the link length, etc. In order to correct the error, we simply applied a factor *1.1* before x and y coordinates. After applying this technique, the error reduced to 2%, which is acceptable.

After the correction of FK, we applied the same technique to IK, leading to a satisfactory result. x and y were almost

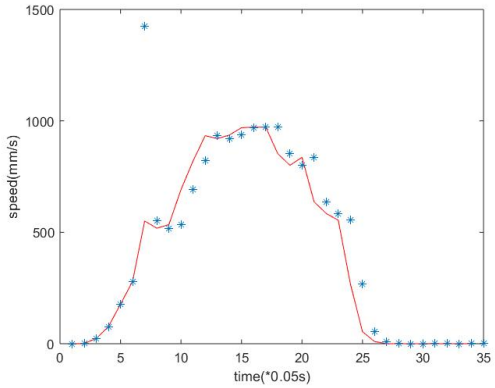

Fig. 12: Endeffector velocity with path smoothing

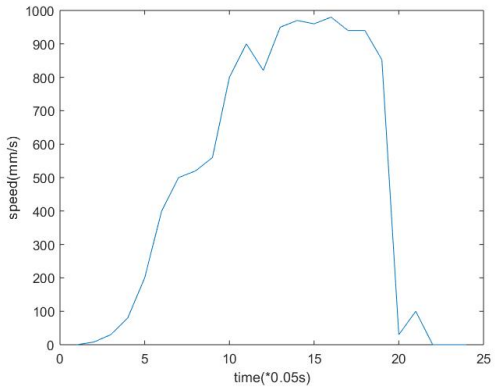

Fig. 13: Endeffector velocity without path smoothing

desired	FK calculated
120, 150	130, 162
100, 120	112, 130
50, 50	56, 57
70, 80	77, 89

Fig. 14: Table for comparison

always correct, while z suffered a bit due to the linearization of depth value from camera. We diminished the error by stacking blocks and adding offsets accordingly. To validate our kinematics, we tested 10 different cases with various testing points. The error was proved to be no more than 2% of the truth value, and our arm was able to grab the block all the time, despite the flaw that the block was not perfectly centered on the gripper.

C. Camera Calibration & Block Detection

Calibration verification and accuracy:

We define the zero Z-coordinate to be on the board. To verify the accuracy of the calibration, first we move the mouse on the board of the image, and compare the accuracy of the calculated world coordinate with the actual measured world coordinate. Second, we place various number of layers of blobs on top of each other on the board. Ideally, even though the layer of blobs increases, when we click the center of the top blob on the image, the resulting calculated world coordinate would only change in Z direction, the world coordinate of the center of the blobs would remain unchanged. We use this criteria to verify our calibration.

As illustrated in Fig. 15 to Fig. 18, we use blobs to help us verify the accuracy of our calibration. In Fig. 15, the X_w, Y_w world coordination of the center of the red marked single layer blob is roughly [150mm, -100mm], and the height of one blob is 38mm. The obtained world coordinate at the center of the blob surface is [149mm, -100mm, 38mm] as shown in Fig. 15. In Fig. 16, the X_w, Y_w world coordination of the center of the green marked 2-layer blob is roughly [150mm, -200mm], and the height of 2 blobs is 76mm. The obtained world coordinate at the center of the blob surface is [154mm, -184mm, 77mm] as shown in Fig. 16. In Fig. 17, the X_w, Y_w world coordination of the center of the red marked single layer blob is roughly [150mm, 150mm], and the height of one blob is 38mm. The obtained world coordinate at the center of the blob surface is [148mm, 147mm, 37mm] as shown in Fig. 17. In Fig. 18, the X_w, Y_w world coordination of the center of the red marked 3-layer blob is roughly [-100mm, 150mm], and the height of one blob is 114mm. The obtained world coordinate at the center of the blob surface is [-100mm, 160mm, 115mm] as shown in Fig. 18. To summarize, we have achieved good accuracy, especially the accuracy in X_w, Y_w position as Z_w increases.

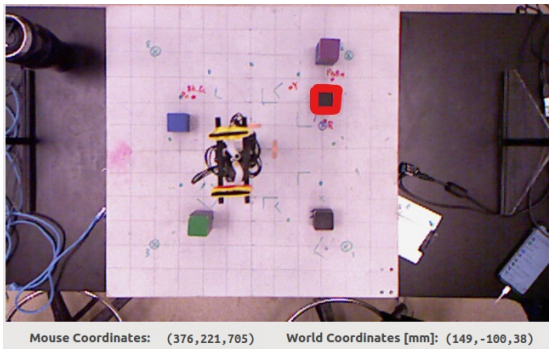

Fig. 15: calibration verification case 1

The performance of our blob detector are demonstrated in Fig. 19 and Fig. 20. We place 25 blobs with different colors on the board as is shown in Fig. 19, and all the blobs are detected with SimpleBlobDetector function as shown in Fig. 20. The detailed results are summarized in Table I and Table II. Finally, the 2D and 3D error VS. location plots are shown in Fig. 21 and Fig. 22, respectively.

D. Autonomous Execution For Final Bits and Pieces

- **Gripper Performance**
  To grab blocks, we tried different open-position and closed-position for the gripper, and find the optimal

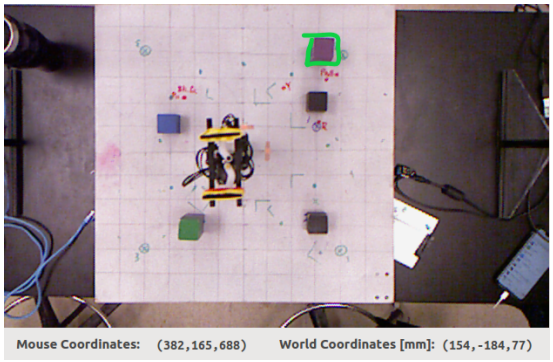

Fig. 16: calibration verification case 2

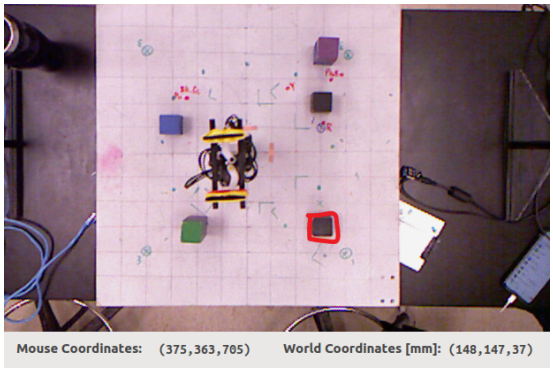

Fig. 17: calibration verification case 3

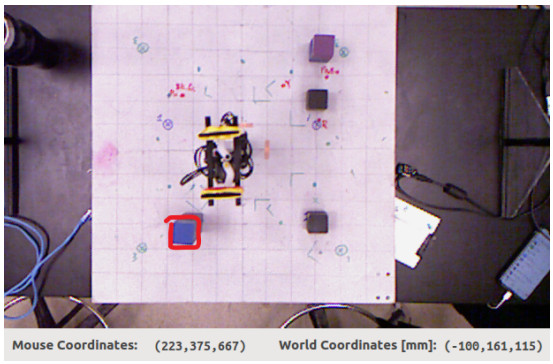

Fig. 18: calibration verification case 4

value which makes the open-length 55.10mm and closed-length 37.50mm. This does behaves well, however, the blocks sometimes slide off from the gripper. Thus we add several elastics and find it works so well that the blocks seldom fall down.

- **Pick and Stack**
  In this event, we did well in the competition that we succeeded to detect three blocks and stack them up at the assigned location. The stack did not fall down. We finished the task, however, there are some imperfectness such that the stack is not a strict vertical line. This means our arm is not so accurate to give us a perfect performance.
- **Wacky Pick and place**
  In this task, we succeeded to pick one block and move

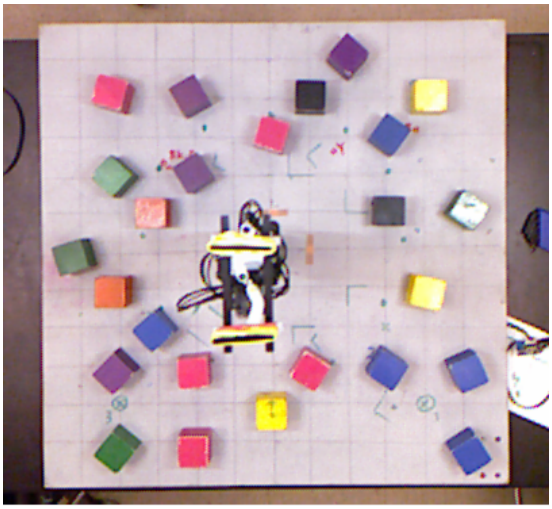

Fig. 19: Actual blobs in image

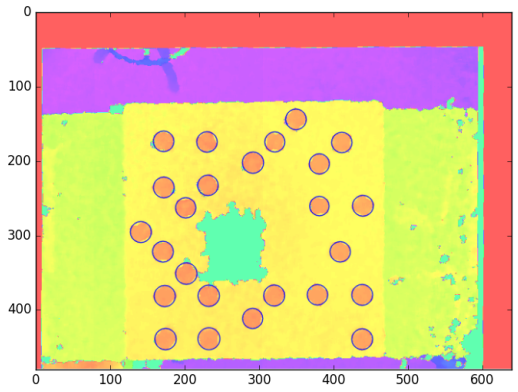

Fig. 20: Blobs detected after SimpleBlobDetector function

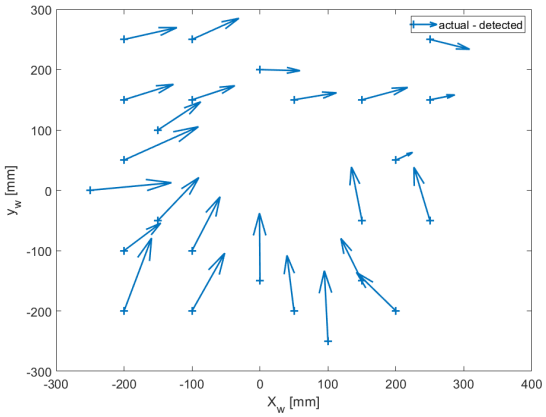

Fig. 21: Error visualization in 2D

it from the initial from to the other one. But it was not always successful. We did trials for many times and finally succeed to finish the task. This means our algorithm is not so accurate. The orientation the block detector generated is not accurate and the fact that it

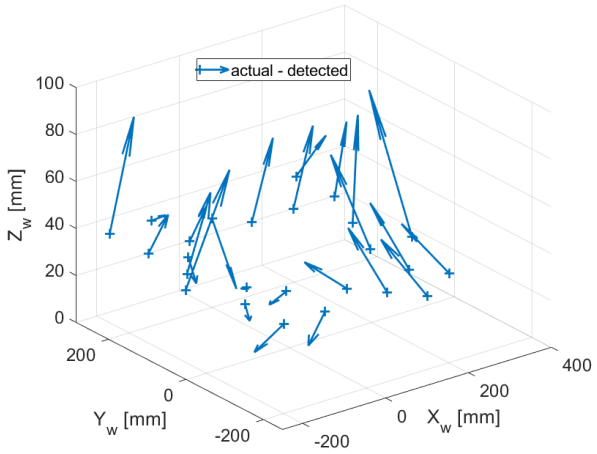

Fig. 22: Error visualization in 3D

often failed to place the block on the destination shows that our kinematics is not accurate. These inaccuracy shows our algorithm should be further developed.

- **Line Them Up**
  In this event, we did part of success to line the blocks in stacks. And, as before, we also tried many times to get the best performance. And the final results shows that the locations that the block detector generates are not so accurate and the kinematics are not accurate either.
- **Stack Them High**
  In this event, we did part of the success, stacked about 5 blocks up. However, as shown above, the generated stack is not a strict vertical line and easy to fall down. This is also about the accuracy of the location and kinematics.
- **Block Mover**
  In this task, we did a success, moved 2 blocks along the required path. But when we tried to move a higher stack, they always fell down.

IV. Discussion about the Possible Improvements

In this lab, we did a success but something should be further improved. For example, the inaccuracy of the kinematics leads to the inaccuracy of the placed blocks' locations. The inaccuracy of the location that the block detector gives also leads to deviation in the performance. This can be partly solved by further explore the trajectory planner and kinematics. However, these deviation can never be removed because there are some hardware and method limit for this project. For example, when we do the calibration, we choose pairs of points generated by clicking in the RGB image and the depth image, which is not an accurate method and the deviation produced by clicking can be enlarged greatly for the calibration process, thus leads to great error. Besides, the servo cannot rotate to the angle that we require, and when the arm arrives at the destination, it would vibrate slightly, which would also generate deviation.

Thus, we can do better in the calibration and kinematics, but we only reduce the deviation instead of removing it.

References

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

[2] S. MALLICK, "Blob detection using opencv (python, c++)," https://www.learnopencv.com/blob-detection-using-opencv-python-c/, 2017.

[3] V. GUPTA, "Color spaces in opencv (c++/ python)," https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/, May 2017.

V. Appendix: Blob Detector Outputs

world coordinate detected (mm) (actual)	color detected (actual)	orientation detected (deg)
[−5.45, 200.16, 41.12] ([0, 200, 38])	yellow (yellow)	−0.0
[244.59, 251.51, 36.51] ([250, 250, 38])	blue (blue)	−30.96
[−106.39, 246.76, 38.39] ([−100, 250, 38])	pink (pink)	−0.0
[−207.20, 248.18, 33.85] ([−200, 250, 38])	green (green)	−54.46
[−206.74, 147.63, 36.99] ([−200, 150, 38])	purple (purple)	−45.0
[246.61, 149.25, 35.24] ([250, 150, 38])	blue (blue)	−26.57
[44.20, 148.93, 35.03] ([50, 150, 38])	pink (pink)	−57.53
[143.77, 148.11, 35.14] ([150, 150, 38])	blue (blue)	−56.31
[−105.83, 147.84, 37.10] ([−100, 150, 38])	pink (pink)	−5.19
[−155.83, 95.67, 39.73] ([−150, 100, 38])	blue (blue)	−45.0
[−210.13, 44.88, 35.72] ([−200, 50, 38])	orange (orange)	−0.0
[197.73, 48.82, 33.88] ([200, 50, 38])	yellow (yellow)	−78.69

TABLE I: Blob detection results

world coordinate detected (mm) (actual)	color detected (actual)	orientation detected (deg)
[−261.08, −1.16, 33.86] ([−250, 0, 38])	green (green)	−18.43
[151.44, −58.20, 34.81] ([150, −50, 38])	black (black)	0.0
[−155.57, −56.59, 38.92] ([−150, −50, 38])	orange (orange)	−0.0
[−205.00, −104.23, 39.31] ([−200, −100, 38])	green (green)	−47.73
[−103.82, −108.26, 39.45] ([−100, −100, 38])	purple (purple)	−33.69
[152.87, −156.50, 35.72] ([150, −150, 38])	blue (blue)	−53.75
[0.07, −160.38, 37.82] ([0, −150, 38])	pink (pink)	−71.57
[205.39, −205.92, 36.22] ([200, −200, 38])	yellow (yellow)	−86.99
[−203.75, −211.18, 40.29] ([−200, −200, 38])	pink (pink)	−74.05
[−104.45, −208.87, 40.37] ([−100, −200, 38])	purple (purple)	−29.05
[100.49, −260.80, 36.62] ([100, −250, 38])	purple (purple)	−53.13
[252.18, −58.16, 32.68] ([250, −50, 38])	green (green)	−75.96
[50.95, −208.53, 36.09] ([50, −200, 38])	black (black)	0.0

TABLE II: Blob detection results, continued