[[1]](#footnote-2)

A Robotic Arm for Cube Detection and Manipulation

Yi Chen, Da Li, and Zhen Wu

*Abstract*— This report presents a six degree-of-freedom (DOF) robotic arm (Rex-arm) with integrated acting, sensing and reasoning functionalities to autonomously grasp, align and stack cubes. A 2 DOF gripper is designed and fabricated by 3-D printing to perform grasping and releasing the cubes. Secondly, a Kinetic sensor (a RGB camera and a depth sensor) is calibrated and used to detect and localize the cubes. The accuracy of the sensing system is demonstrated to be 2.3 mm in RMSE. A motion planning algorithm incorporating solving inverse kinematic and generating intermediate way points is developed to perform pick and place of cubes. Last, a finite state machine is implemented to integrate each module to perform events specified user.

# Introduction

Robotic manipulators are electronically controlled mechanism, consisting of multiple segments that perform tasks by interacting with the environment. They have been widely adopted to perform industry specific tasks such as picking and placing objects [1]. In this report, we present a 6 Degree-of-Freedom (DOF) robotic arm to autonomously detect, pick, move, align and/or stack cubes. To realize the goal, we decompose the task into four major milestones: gripper design, sensing, kinematics and planning with finite state machine.

The report is organized as follows: Section II describes the overview of the experimental setup used in this study. Section III describes the method studied for path smoothing and its performance evaluation. Section IV describes the final design of the gripper, design trade-offs and improvements made during the course of study. Section V describes the method we adapted to perform camera calibration and the algorithm implemented to detect and locate cubes. Section VI describes the derivation of forward and inverse kinematics and Section VII describes the algorithm for motion planning and implementation of FSM. Finally, The results of competition are showed and discussed in Section VIII and IX respectively.

# Methodology

## Path Planning

In this part, we developed a planner to make robot move to the desired place without collision. The planner is a waypoint generator based on A\* search algorithm. The waypoints are also decided by using the real-time coordinates from odemetry and optitrack.

### Description of algorithm: In order to complete the task 4, we used A\* search algorithm to find the path. Comparing with Dijkstra's algorithm, A\* search algorithm is much faster by usi ng best-first search to speed things up.

In this competition, we conisiderred our robot as a point and all obstacles as cirlces with radius. With the range of radius, the collision will happen. So in this task, we set the radius of obstacles as half of the maximum width of the robot and the radius values are shown in Table xx.

With the configuration space of the world, an A\* search algorithm was used to construct a tree of paths starting from start node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node. At each iteration of its main loop, A\* selectted the path by using approximate total path cost, which is to minimize an evaluation function:

Where n is the last node/point on the path, is the cost of the path from the start node to n, and is a heuristic function that estimates cost from n to the goal. is the estimated total cost of path through n to goal.

In this task, we define as the Euclidean distance from the position of last node n to goal shown in Eq. xx and this heuristic is admissable since it never overestimated the cost to reach the goal.

Where () and () are the coordinates of current and goal node, respectly.

With this heauristic, we can calculate estimated total cost of path and use priority queue to perform the repeated selection of minimum estimated cost nodes to expand. So at each iteration, the node with the lowest is removed from the queue, and the and values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower value than any nodes in the queue or until the queue is empty. The value of the goal is then the length of the shortest path, since at the goal is zero in an admissible heuristic. So The A\* search algorithm for reaching a single destination is summarized in Algorithm 1.

|  |
| --- |
| **Algorithm 1** Algorithm of path planner based on A\* search algorithm |
| **while** () &&  dequeue:    **for** each *nbr* in not visited(adjacent())  **if** !*collision(nbr)*  enqueue: *nbr to*  **if**        e**nd if**  e**nd for**  e**nd while** |

### Gate Waypoint Placing: Since we are given the position of left and right gates, the direction of passing through the gate is fixed. So that we can define two waypoints placed on both sides of the gate as entering point and exiting point.

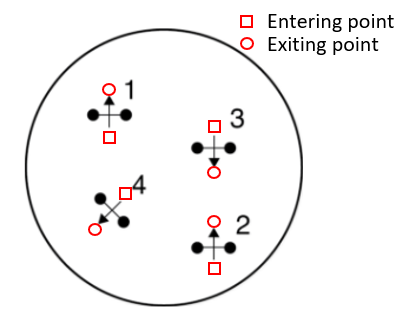


Fig. xx: An example of gate configuration. Square represent entering point and circle is exiting point.

The entering and exiting waypoints are calculated by using the coordinates of left and right gate shown in following equations:

Where is the maximum width of the robot and () and () are entering and exiting waypoints we will set.

### Final implementation: The robot moves from the entering point to exiting point to pass through the gate. Once the robot reaches the exiting point, the gate will be open, which will be not considered as obstacles in the rest of path-finding. Then we will use A\* search algorithm to find a path from last exiting point to next entering point. So in the competition, there are 4 gates in total, So 4 entering points and 4 exiting points are generated together with starting point and goal point. And between each adjacent waypoint, we used RTR planner to go in a straight line. We coded A\* search algorithm in Python to generated waypoints data file for the competition and the path is shown in the result.

# results

## Path planning

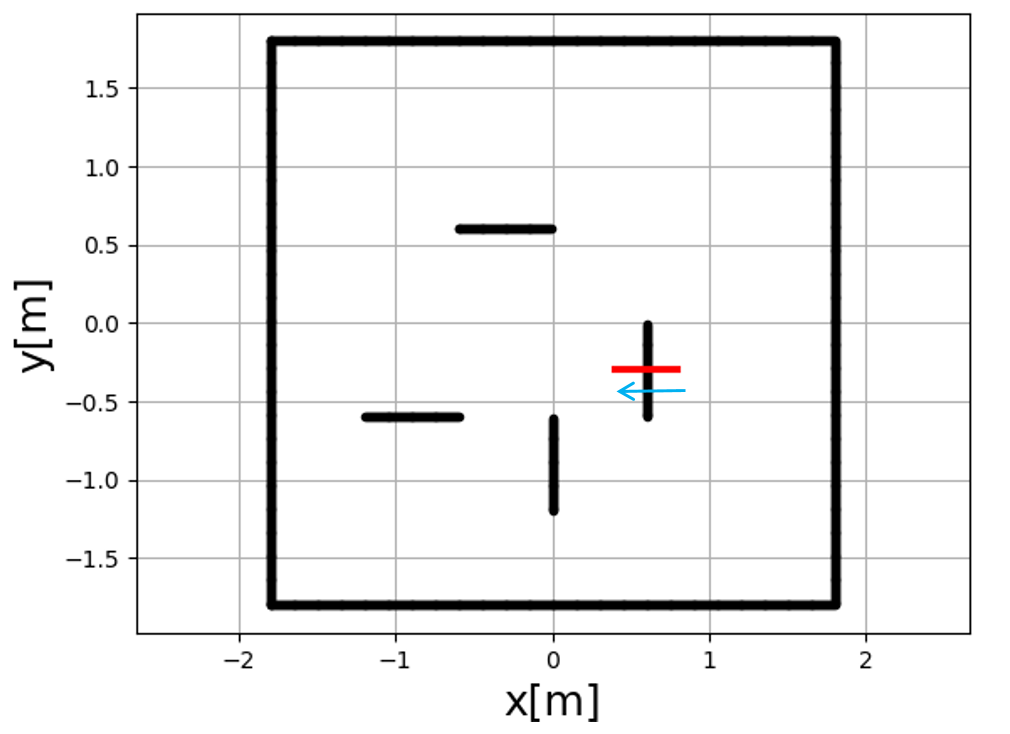
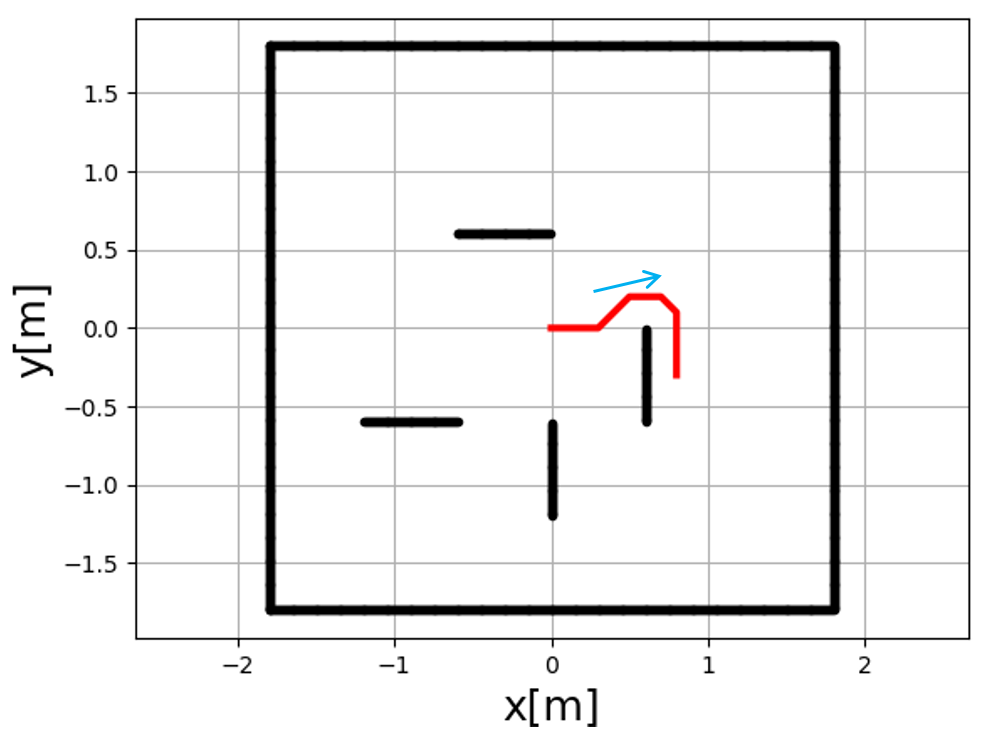
Based on the gate positions obtained from optitrack, we firstly run a Python simulation to generate the waypoints in the whole path. The gate positions are shown in following table.

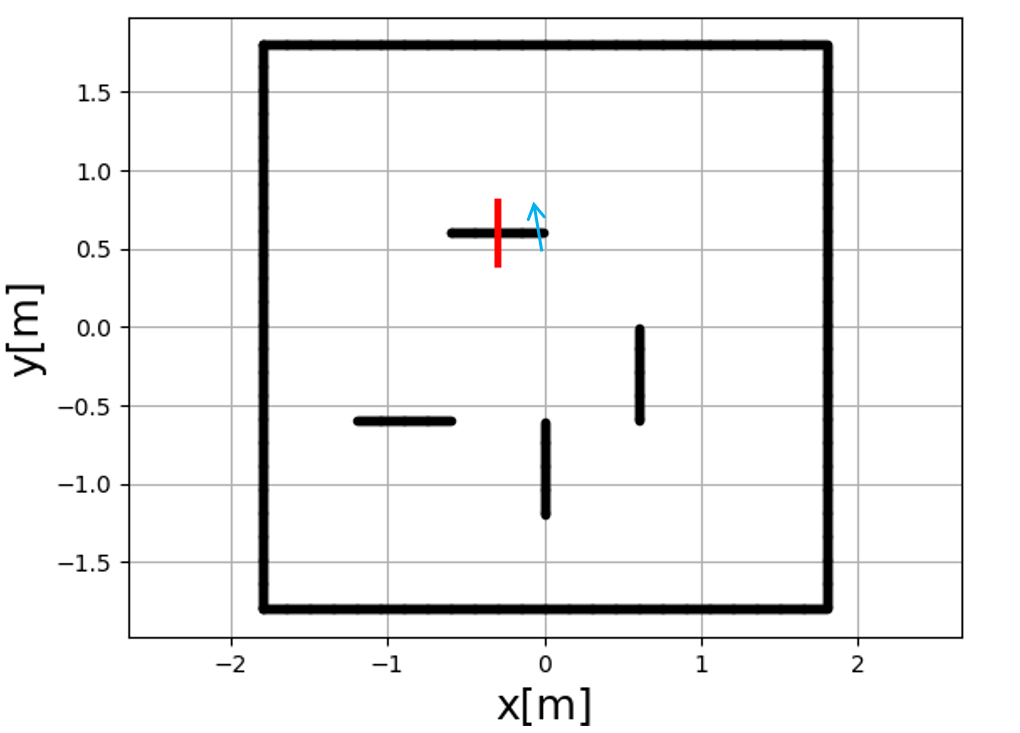
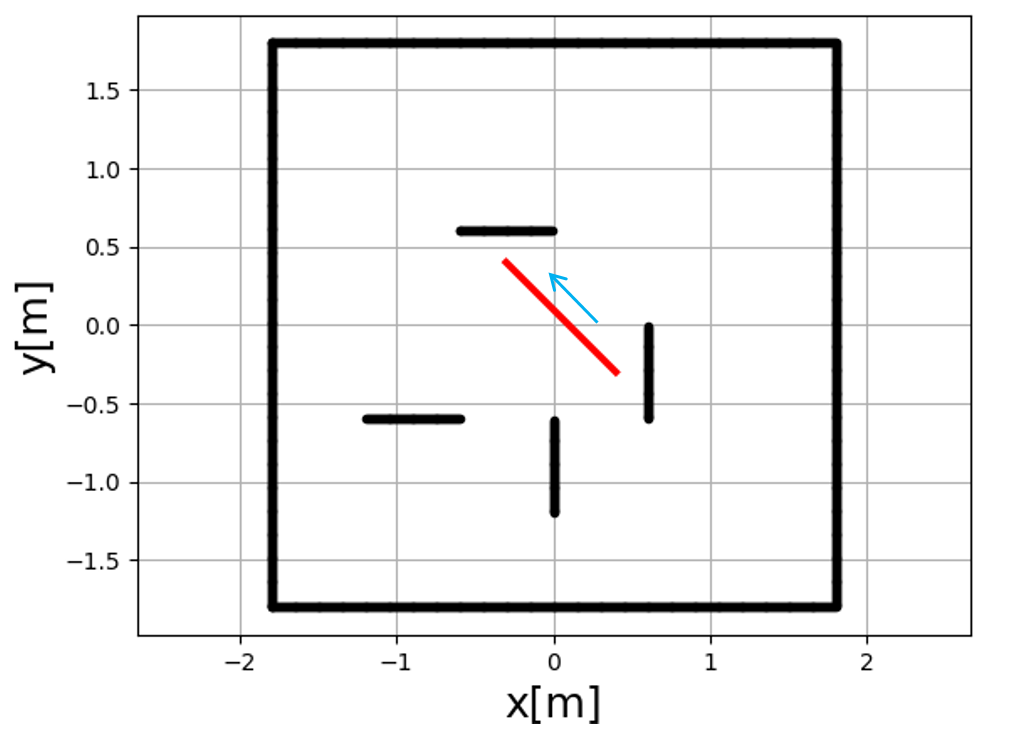
TABLE xx

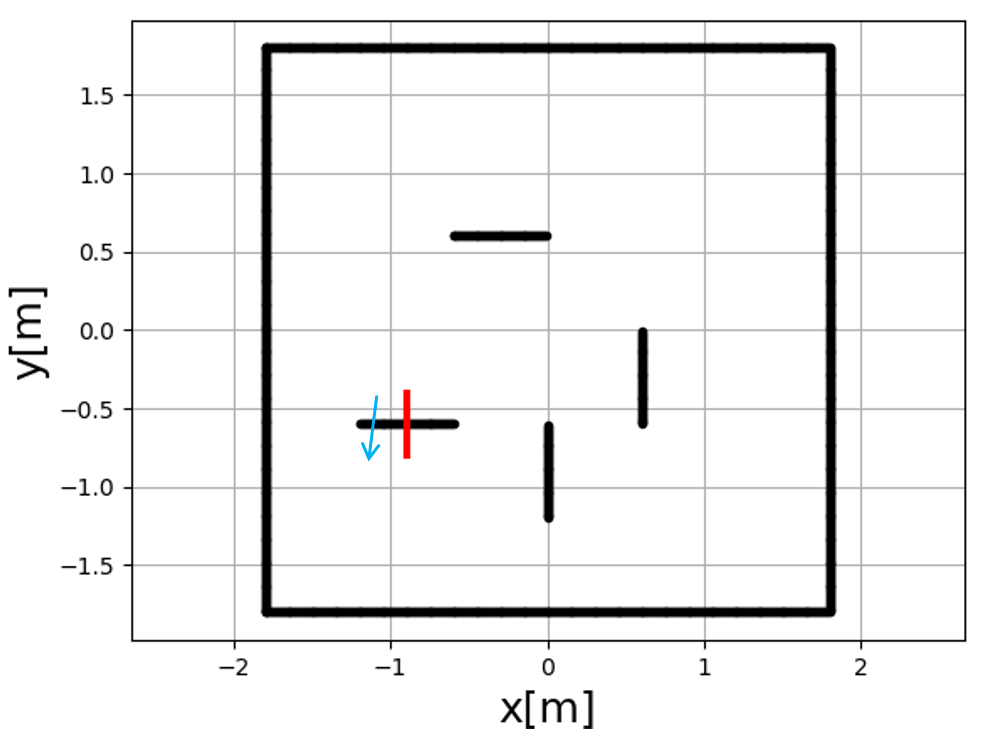
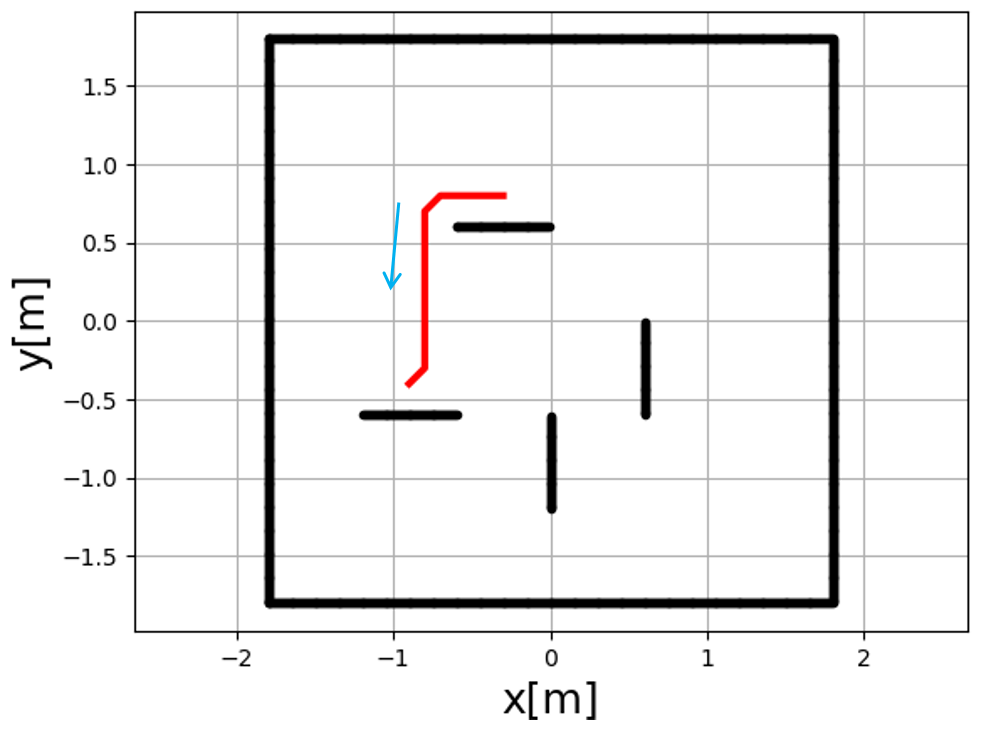
The position of each gates

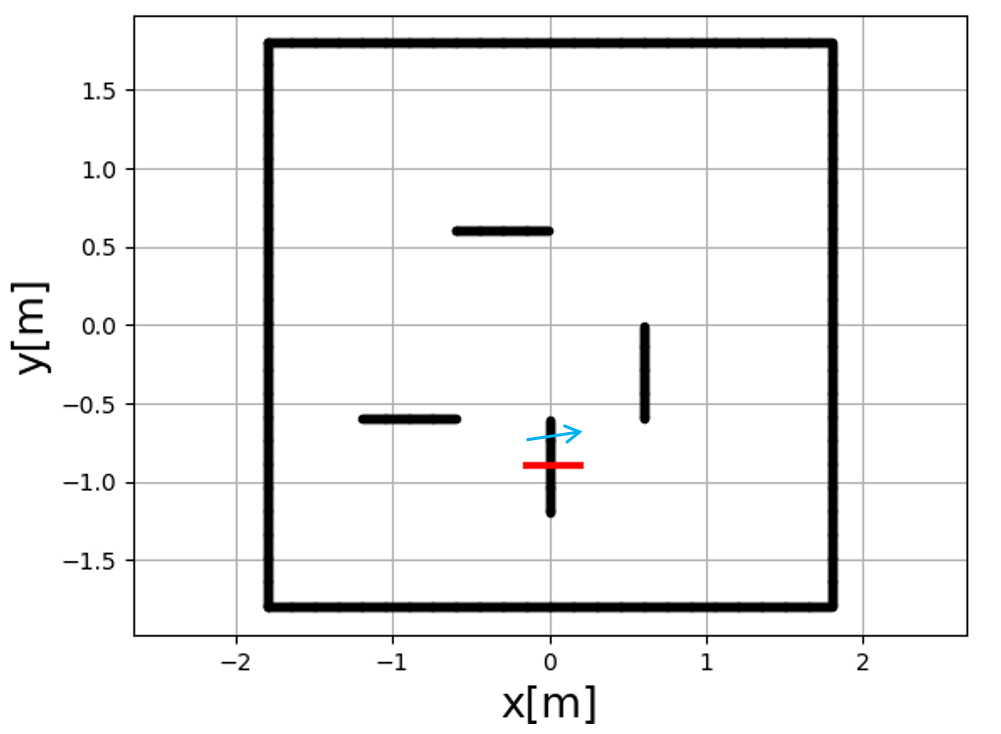
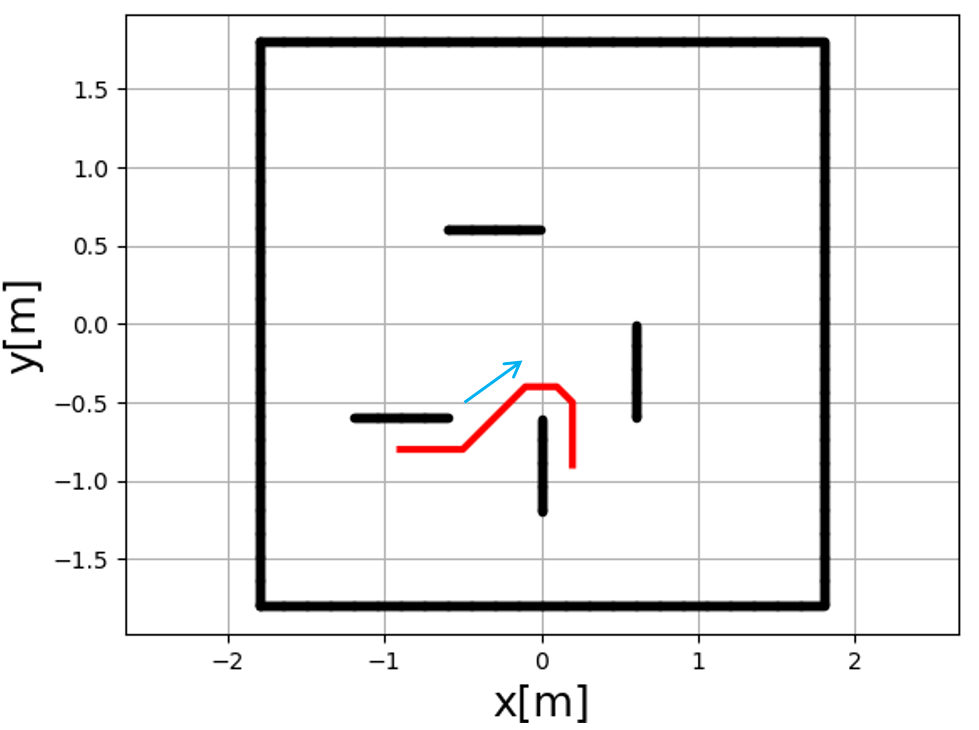
|  |  |  |
| --- | --- | --- |
| # of gate | Left [m] | Right [m] |
| 1 | (0.062, -0.064) | (0.062, -0.001) |
| 2 | (-0.061, 0.065) | (0.002, 0.062) |
| 3 | (-0.064, -0.061) | (-0.121, -0.063) |
| 4 | (0.001, -0.123) | (0.001, -0.061) |

The result of path is shown in Fig. xx.









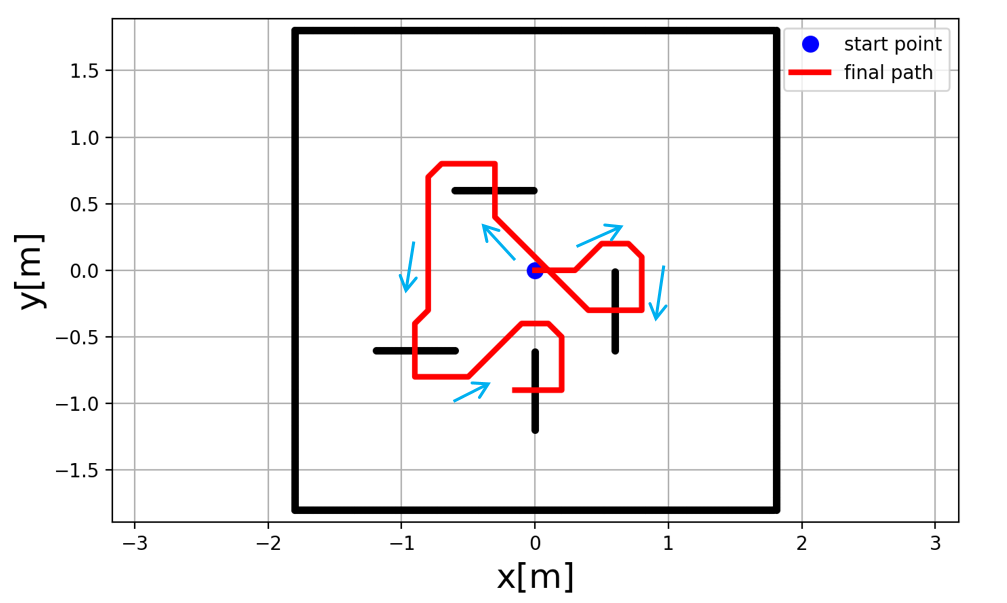


Fig. xx: 8 pieces of partial paths and the whole path

Despite the accurate prediction on colors using previous images [9], the threshold for color detection fails for most of the cases during contest. In fact, given the threshold, it predicts most of the color to be violet, as shown by the confusion matrix Figure 12. To generate the confusion matrixed, we placed each cube with a certain color on the board positions 1 to 8 as labeled in Figure 7(a) and took 5 measurements for each position. The probability was then calculated using the same method as before: the number of cubes for detected color divided by the total cube number for each color. We believe that the accuracy drop may be caused by the lightening condition change from our working station to the station on which sample images were taken. Therefore, to improve the detection accuracy, we propose to use the images captured from the stations one plans to work on under various lighting environments. Another potential alternative is to train a more robust color detection algorithm using Neural Network [11].



Figure 12. Confusion matrix for color detection during contest

## Overall Performance

In the final competition, we managed to finish the event 1 and event 2 with only one trial. We did not finish the event 3 and event 4 due to the issue of insufficient accuracy of color detection for the block. For event 5, we managed to pile 2 levels of pyramids in our internal test, but did not manage to tune the code robustly enough to accomplish higher level of pyramid building.

## Discussion on improvements

For future works, several action items can be done to potentially improve the performance:

1) The transition way points can be more customized for each event, so that the arm may move more efficiently. For example, in event 1 all the blocks are at the first level of height, therefore the transition points can be set the top of the block before picking. In this case, the arm does not have to move to the vertical position every time.

2) More testing and tuning for the color detection need to be done to produce a robust result to support the entire execution of event 3 and event 4.

# Conclusion

References

|  |  |
| --- | --- |
| [1] | ROB550. (2018) Balancebot description. |
| [2] | Borenstein, Johann, and Liqiang Feng. "Gyrodometry: A new method for combining data from gyros and odometry in mobile robots." Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on. Vol. 1. IEEE, 1996. |
| [3] | Borenstein, Johann, and Liqiang Feng. "Measurement and correction of systematic odometry errors in mobile robots." IEEE Transactions on robotics and automation 12.6 (1996): 869-880. |
| [4] | Modeling a simple DC motor, P. Gaskell, 2018 |
| [5] | Lerner, Jürgen, Dorothea Wagner, and Katharina Zweig, eds. Algorithmics of large and complex networks: design, analysis, and simulation. Vol. 5515. Springer, 2009. |

# Appendix A: camera calibration

## A.1 Depth function identification

Raw data used to identify the depth function depicted in Figure 6.

TABLE A.1

BOM

|  |  |
| --- | --- |
| Part name | # of parts |
| Beaglebone Green | 1 |
| Mobile Robotics Cape | 1 |
| 3 cell, 1500mAh Lithium Polymer battery | 1 |
| battery monitor | 1 |
| 20.4:1 Metal Gearmotor 25Dx50L mm MP 12V with 48 CPR Encoder | 2 |
| DRV8801 Single Brushed DC Motor Driver Carrier | 1 |
| MPU9250 IMU | 1 |
| DSM Satellite receiver | 1 |
| Acrylic board | 1 |
| Metal pillar | 6 |
| Robot wheel | 2 |
| M3 screws | 40 |

## A.2 Depth and RGB camera association

Setup image for depth and RGB camera association, dots are the positions used for calculation and detailed pixels values in RGB and depth camera are documented in Table A.2 below.



TABLE A.2

data for depth and RGB camera association

|  |  |  |
| --- | --- | --- |
| # of points | RGB camera  (x, y) | Depth camera  (x, y) |
| 1 | (144, 93) | (147, 70) |
| 2 | (140, 448) | (147, 456) |
| 3 | (204, 211) | (211, 196) |
| 4 | (203, 270) | (210, 260) |
| 5 | (201, 325) | (210, 326) |
| 6 | (321, 155) | (340, 133) |
| 7 | (318, 387) | (340, 389) |
| 8 | (435, 211) | (468, 197) |
| 9 | (434, 268) | (470, 259) |
| 10 | (436, 330) | (471, 325) |
| 11 | (498, 93) | (530, 69) |
| 12 | (499, 450) | (538, 456) |

## A.3 Extrinsic matrix calibration

TABLE A.3

data for extrinsic matrix calibration

|  |  |  |  |
| --- | --- | --- | --- |
| # of points | World coord.  (x, y, z)[mm] | # of points | World coord.  (x, y, z)[mm] |
| 1 | (200, -200, 0) | 9 | (100, -100, 152) |
| 2 | (0, -200, 38) | 10 | (0, -100, 38) |
| 3 | (-200, -200, 76) | 11 | (-100, -100, 76) |
| 4 | (-200, 0, 114) | 12 | (-100, 0, 114) |
| 5 | (-200, 200, 0) | 13 | (-100, 100, 152) |
| 6 | (0, 200, 38) | 14 | (0, 100, 38) |
| 7 | (200, 200, 76) | 15 | (100, 100, 76) |
| 8 | (200, 0, 114) | 16 | (100, 0, 114) |

Algorithm for extrinsic matrix calibration:

With both intrinsic matrix and depth function, Eq. (1) can be written as

(A.1)

where the only unknown is the extrinsic matrix that describes the coordinate transformation from 3D world frame () to 3D camera frame (). Hence, the dimensions for and matrices in Eq. (A.1) are 3 by 3 and 3 by 1 respectively. This implies that there are a total of 12 parameters to be estimated to reconstruct the extrinsic matrix. For a pair of known correspondence, we have three equations

(A.2)

where the superscript indicates the pair. For parameter estimation, Eq.(3) can be rearranged as

(A.3)

where consists of the parameters in extrinsic matrix. With multiple pairs of known correspondences [7], we can expand Eq.( A.3) as

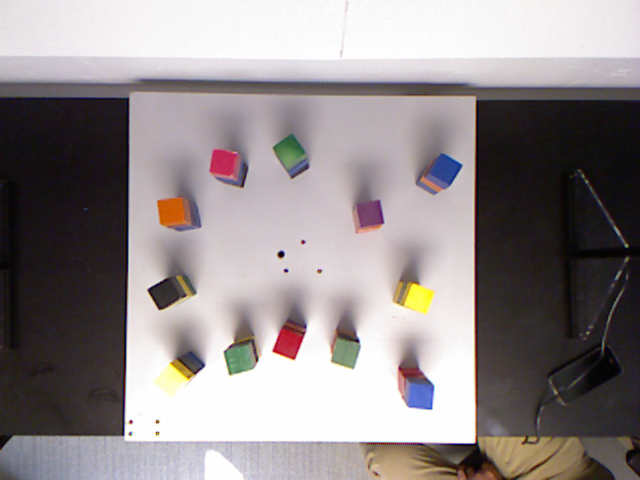
and parameter can be determined using least square estimation , where

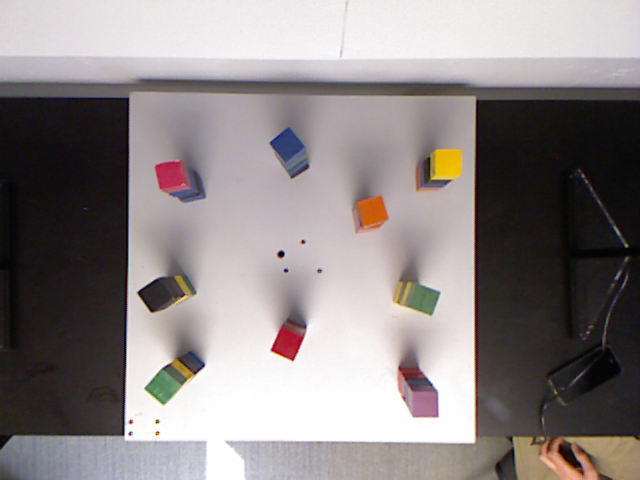
# Appendix block detection

Training image



Validation images





Testing images

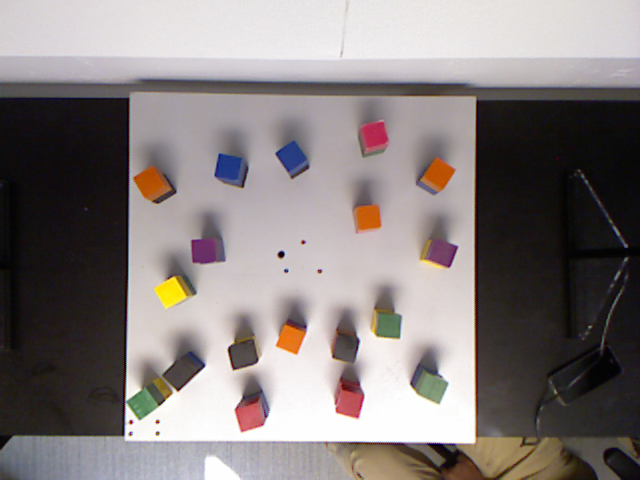




TABLE B.1

color thresholds

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Color | H | S | V | L |
| Black | (0, 255) | (0, 80) | (0, 90) | (0, 100) |
| Red | (150, 200) | (100, 200) | (0, 255) | (0, 100) |
| Orange | (10, 40) | (0, 255) | (0, 255) | (0, 255) |
| Yellow | (20, 30) | (0, 255) | (0, 255) | (0, 255) |
| Green | (40, 100) | (0, 255) | (0, 255) | (0, 255) |
| Blue | (80, 130) | (0, 255) | (0, 255) | (0, 255) |
| Violet | (100, 140) | (0, 255) | (0, 255) | (0, 255) |
| Pink | (120, 255) | (0, 255) | (0, 255) | (120, 255) |
| Color | R | G | B |  |
| Black | (0, 255) | (0, 255) | (0, 255) |  |
| Red | (0, 255) | (0, 100) | (0, 100) |  |
| Orange | (125, 255) | (60, 140) | (0, 50) |  |
| Yellow | (0, 255) | (150, 255) | (0, 255) |  |
| Green | (0, 150) | (50, 255) | (0, 255) |  |
| Blue | (10, 100) | (0, 255) | (100, 255) |  |
| Violet | (100, 255) | (0, 255) | (100, 255) |  |
| Pink | (180, 255) | (0, 150) | (0, 255) |  |

# Appendix c: finite state machine design

## Detailed finite state machine

|  |
| --- |
| **Run\_state\_machine(event\_num):** n\_trial = 0 **Detecting all the blocks on the board:** Generate a color\_list, loc\_list  if no block detected:  to state 8  else:  to state 2 **Analyzing board situation based on event\_num:** According to rules specified by event, identify the blocks need to be picked  If no block need to be picked or n\_trial > n\_max :  to state 8  else:  generate color\_list\_pick, loc\_list\_pick from color\_list, loc\_list **Generating target locations, assign orders:** For i = 1 to N in loc\_list\_pick:  compute loc\_list\_place[i]  Reorder loc\_list\_pick, loc\_list\_place according to its color info (only for event 3 and 4), to state 4  For i = 1 to N in loc\_list\_pick: **Compute solution of inverse kinematic:** Compute solution of inverse kinematic given loc\_list\_pick[i], loc\_list\_place[i]:  If no solution exists:  to state 7  else:  Compute angle\_pick, angle\_place then to state 5 **Generating way points:** Given angle\_pick, angle\_place, generates angle\_transition for the arm, then to state 6 **Move one block to target location:** Given the way points, perform sequential movements on the arm and the gripper to pick and place the block **Checking if all blocks have been moved:** If all blocks have been moved:  n\_trial += 1  to state 1  else:  to state 4  endfor **Terminate the event and reset arm position:** Close gripper and reset arm position to vertical pose  Return status of the event |

## Detailed Motion Sequence

For simplicity, the motors are abbreviated as: B (base), S (shoulder), E (elbow), W (wrist), G1 (grip1), G2(grip2). We define the following motion as:

Open gripper: G2 = 90

Close gripper: G2 = 0

Rotate gripper: G1 = 15 or -75

Lift to safety: move S 50 towards 0 direction

Move to pick: move the arm in sequence of B->E->S->W

Move to place: move the arm in sequence of B->S->E->W

Move at same: move B, S, E, W at same time to target

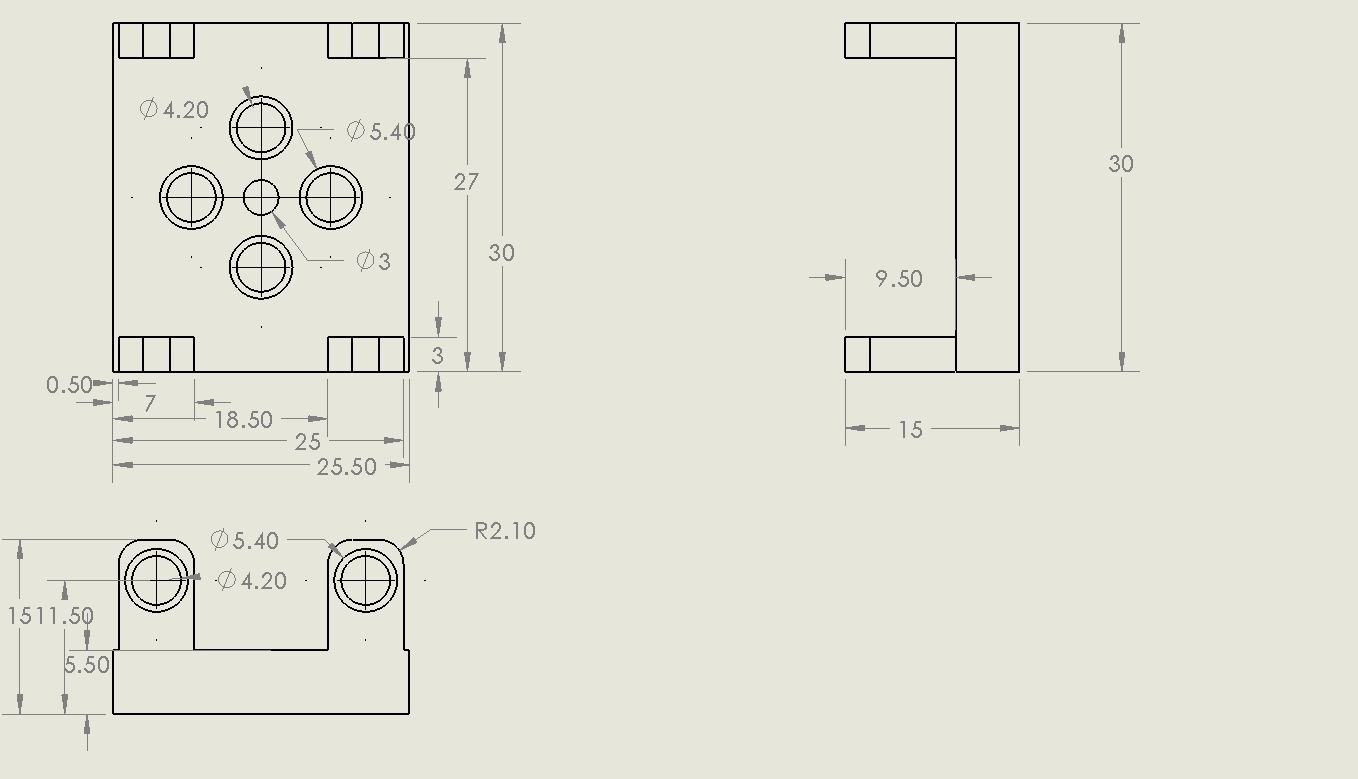
For event 1 to event 4, the motion plan is set to be: open gripper -> move to pick -> close gripper -> lift to safety -> move to transition -> move to place -> open gripper -> lift to safety -> move to transition

For event 5, the motion plan is set to be: open gripper -> move to pick -> close gripper -> lift to safety -> move to transition -> rotate gripper -> move to place -> open gripper -> move at same to transition -> move to transition

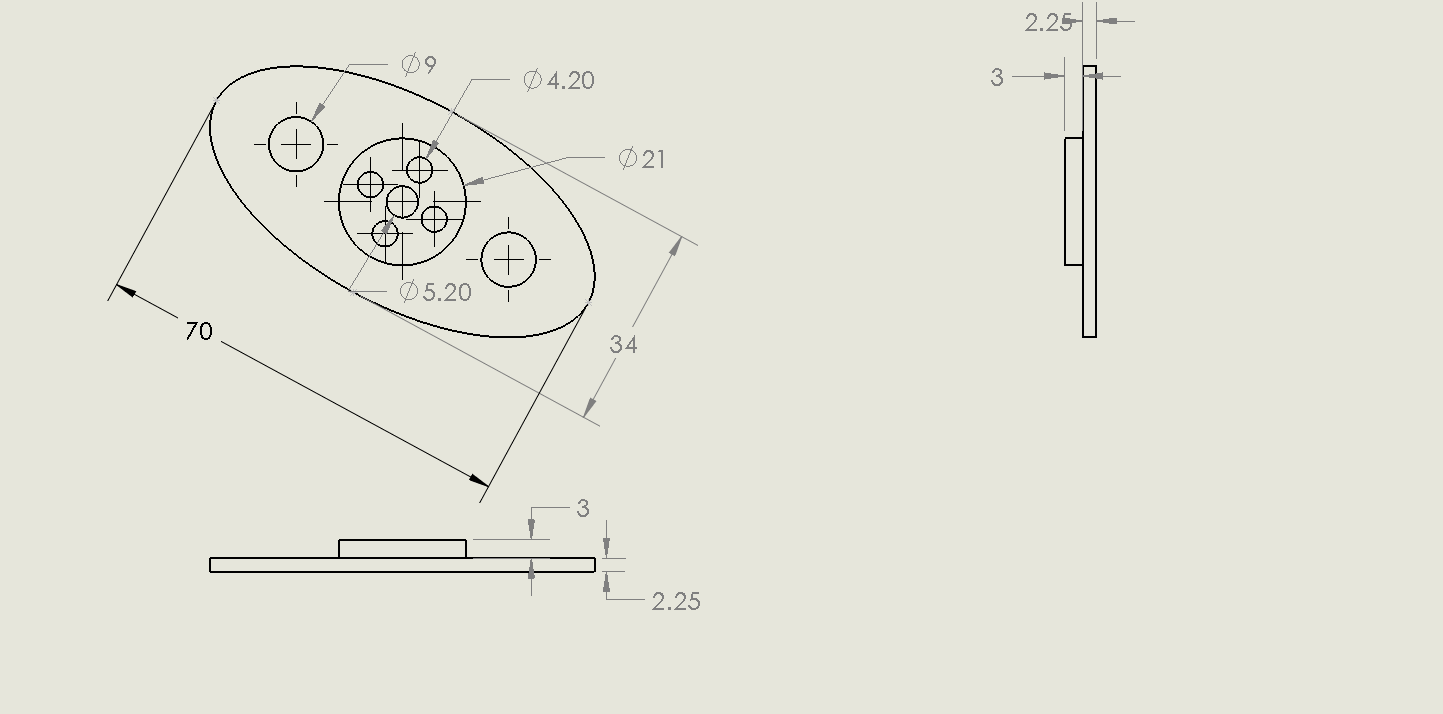
The designed motion plan was able to accomplish pick and place of block without generating interference with other objects during the experiments.

# Appendix d: cad drawing

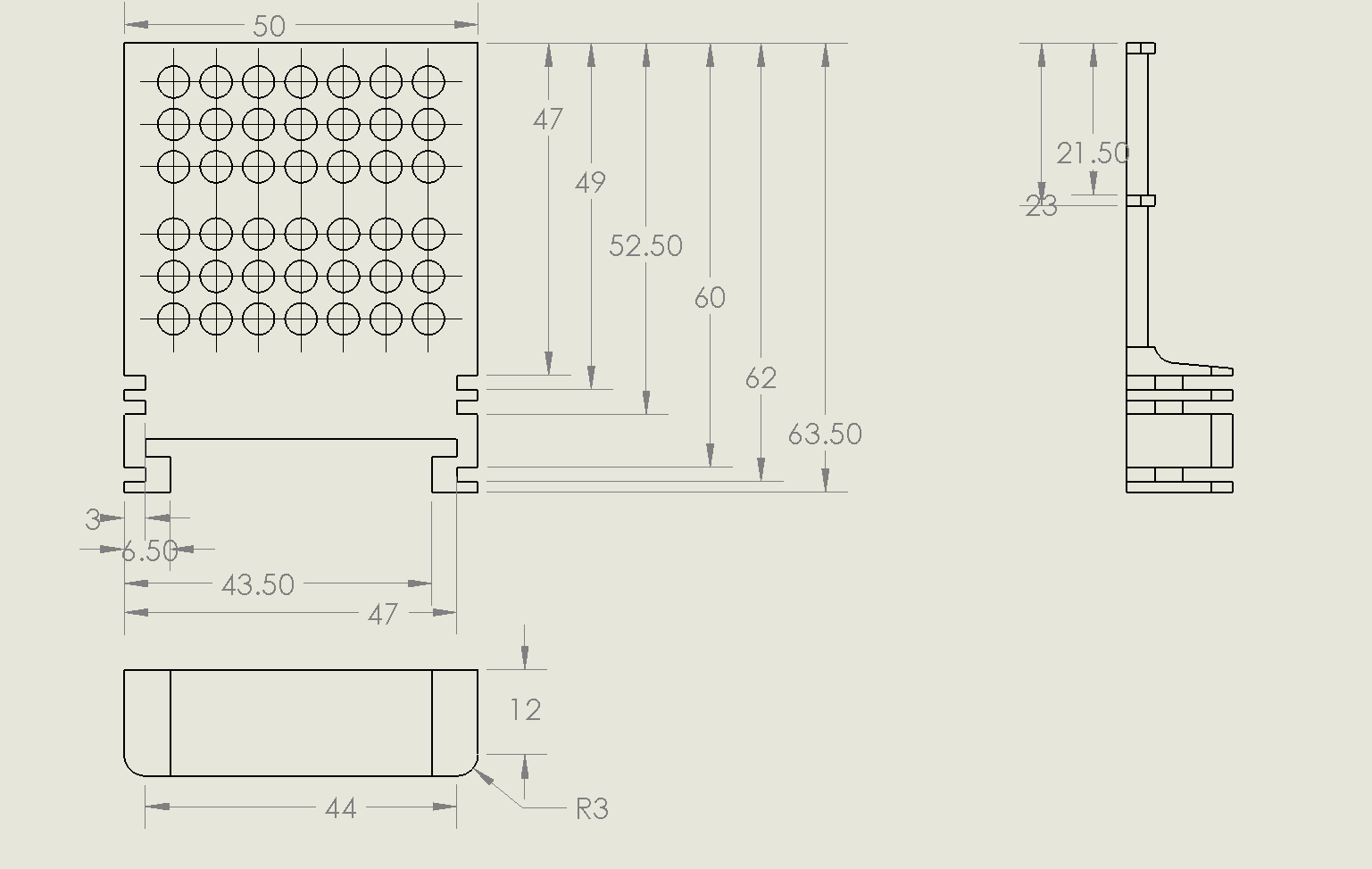
Connector



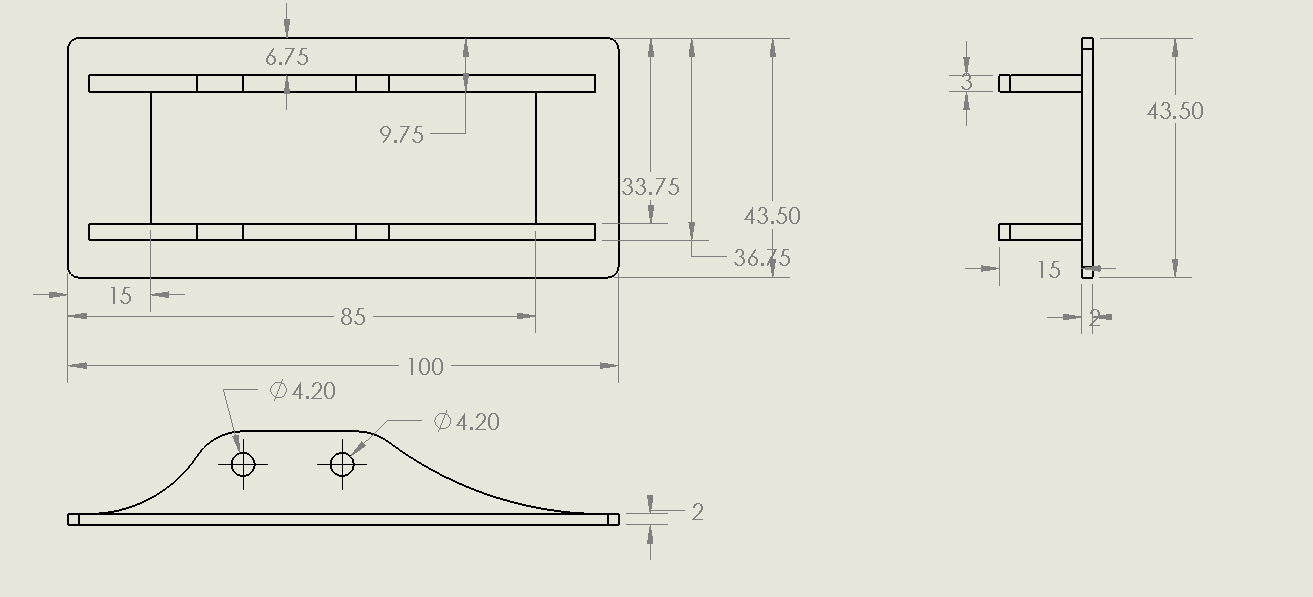
Ellipse with spacer



Gripper hand



Rail



1. [↑](#footnote-ref-2)