[[1]](#footnote-2)

Two-wheel Balanced Robot

Weilun Peng, Da Li, and Jia Shi

*Abstract* — We proposed a methodology of automating a two-wheeled self-balanced robot to search and follow a path with a given destination in a constrained arena. We firstly designed PID controllers to balance and move our robot without falling. Then we developed odometric dead-reckoning to estimate the position of our robot. For motion control, a “Rotate, Translate, Rotate” (RTR) planner is implemented to drive the robot along a determined path. Finally, We implemented A\* algorithm for robot to search shortest path to a target location. All these functions can make robot automatically find and follow a path in a maze.

**Keywords: PID control, dead reckoning, RTR planner, path planning**

# Introduction

In this lab, we are tasked to build a two-wheeled robot, which is capable of performing the following four tasks: 1) self-balancing through PID controllers of speed, angle and position; 2) manually …3) autonomously navigate through four predefined gates on the arena.

This report is organized as follows: Section II describes the methodology including the motor model, PID controllers, odometry, and path planinng. Section III presents the result and discussion. Section IV presents the major conclusions from this project.

# Methodology

## Motor Model

The motor parameters include motor resistance measured directly from the multimeter; motor current and voltage measured directly from the multimeter (under 0.5. 0.7, and 1.0 PWM); no load speed calculated using eq. (1); motor constant calculated using eq. (2); stall torque calculated using eq. (3). Table 1 shows the summary of motor parameters.

(1)

Where is the wheel encoder measurement; is the wheel encoder resolution; is the gear ratio; is the time elapse between two wheel encoder measurements.

(2)

(3)

Table 1. Motor Parameters

|  |  |  |
| --- | --- | --- |
|  | Left | Right |
| Motor resistance (ohm) | 4.7 | 5.5 |
| Motor constant | 0.283 | 0.278 |
| No load speed (rad/s) | 39.85 | 40.42 |
| Stall torque () | 0.67 | 0.55 |

## Controllers

Theoretically, the BalanceBot has an equilibrium pitch angle which points to upright, but this configuration is an unstable system. Any input or response to the system will bring it away from the equilibrium pitch angle but the system does not have the ability to take itself back to the equilibrium pitch angle. Therefore, we designed and implemented a pitch angle controller, a linear velocity controller, and a turning velocity controller to keep the BalanceBot system at the equilibrium pitch angle with a desired linear and angular velocity. Also, position controller and heading controller were implemented to move the BalanceBot to the target position with a desired pose

The block diagram of the cascade controller system are shown in Fig. 1. All five controllers in this diagram are PID controllers. The proportional gain, integral gain, and derivative gain will be reported in table ? (Maybe in the appendix?). To remove the effect of noisy signal, the derivative terms in all five controllers were filtered by a first order low-pass filter with a cut-off frequency of 40 Hz.

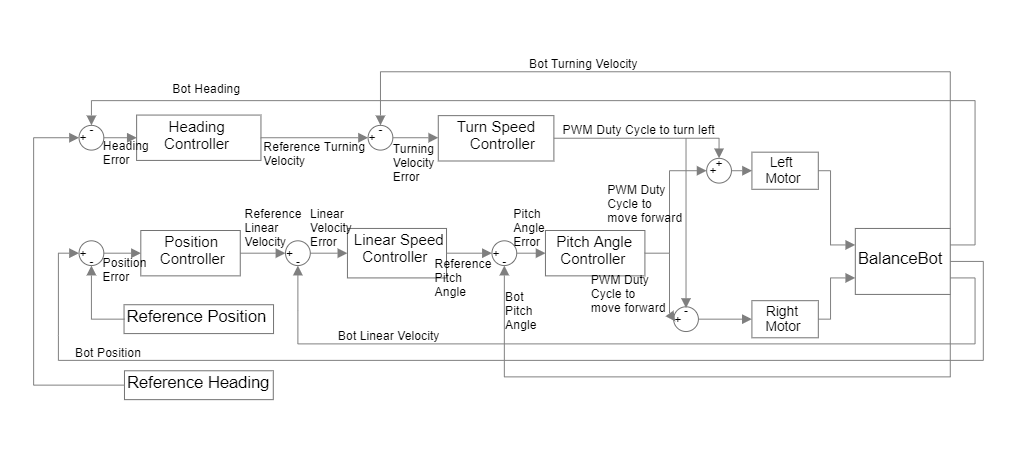


Fig. 1 Block diagram of the cascade controller system

To tune the PID controllers, we first adjusted Kp to a value at which oscillation happened. Then we decreased Kp to a value that oscillation was just removed. Kd was the second gain we tried to tune. We tried to set Kd the same value as Kp first. Then we noticed that there were oscillations and the BalanceBot started to tremble. Kd was decreased to remove these phenomena. At last, KI was added if the BalanceBot always drifted to one directed. By applying suitable KI, the error could be reduced. In our system, KI was necessary for only one controller to reduce the error.

As shown in Fig. 2, the pitch angle is defined as the angle between the symmetry axis of the BalanceBot and the upright. The heading angle is defined as the angle that the BalanceBot is away from the initial facing direction by how much.

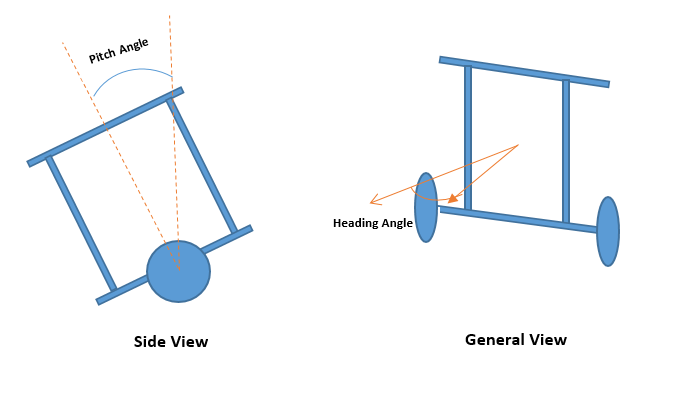


Fig. 2. Simplified model of BalanceBot

### Pitch Angle Controller: The Pitch Angle Controller took the reference pitch angle and the measured pitch angle from IMU. Since the pitch angle from IMU has a nonzero value when the BalanceBot was initialized at a upright state. We waited 15 seconds before running the BalanceBot to calibrate the IMU. Then we picked the first three pitch angle readings from the IMU and used their average as the pitch angle offset of the current operation. We found that the raw pitch angle reading was not continuous at the upright position. Hence, we subtracted 2π when the reading was greater than 0, which made the pitch angle change continuously and symmetric when the BalanceBot bent forward and backward. The Pitch Angle Controller took error which was generated by subtracting the real pitch angle from the reference pitch angle as input and gave the duty cycle of PWM as output to control the motors to keep the BalanceBot at balance position during transition, rotation or standing. The proportional, integral, and derivative parameters will be reported in Table 2.

### Linear Velocity Controller: The Linear Velocity Controller took the reference linear velocity and the actual velocity that the BalanceBot is moving forward with. The actual linear velocity was from the readings of encoders of left and right motors. Before each update, the encoders were reset to 0, which made the readings of encoders the actual values that changed during the period. To compute the linear velocity, we used the equation:

To calculate left and right speeds in RPM. Then the equation:

was used to calculate the linear velocity of the BalanceBot in m/s. Then it was subtracted from the desired linear velocity, which produced the error as the input for the Linear Velocity Controller. At last, the Linear Velocity Controller gave out the desired pitch angle as output. The proportional, integral, and derivative parameters will be reported in Table 2.

### Turning Velocity Controller: Similarly, the Turning Velocity Controller took the same logic of implementation as the Linear Velocity Controller. It took the reference turning velocity and the actual angular velocity that the BalanceBot was turning to the left. The left and right motor speeds were generated through the same equation above. The calculation of turning speed (in rad/s) was by the equation:

Then, the difference between desired turning velocity and the actual value was taken as the input of Turning Velocity Controller and it gave the PWM duty cycle as output. This duty cycle could control the BalanceBot to turn at desired angular velocity to the left if we subtracted it from the duty cycle of left motor and added it to the duty cycle of right motor. When the output was positive, it controlled the BalanceBot to turn to the left. When it was negative, the BalanceBot was controlled to the right. The proportional, integral, and derivative parameters will be reported in Table 2.

### Position Controller: The Position Controller controlled the BalanceBot to stay or move to the desired position. It took the distance from the current position to the desired position as input and generated the desired linear velocity (in m/s) as output. The input distance (in m) was calculated by the equation:

The proportional, integral, and derivative parameters will be reported in Table 2.

### Heading Controller: The Heading Controller took the difference between the desired heading and the current real heading to control the BalanceBot to turn to the desired heading angle. The desired heading angle is computed by the equation:

The current heading angle was obtained from odometry model. To clamp the heading angle difference and make it continuous when the BalanceBot turned backwards, we had come up with a strategy to handle the situation that if the absolute value of the angle difference is greater than π. When the value of the angle difference is greater than π, we made the angle difference negative and plus π to it. When the value of the angle difference is less than –π, we made it to positive and subtracted π from it. Also, we had the same strategy to handle the actual heading angles. After this process, the result from actual heading angles would be subtracted from the difference of heading angle, which would prevent the difference of heading angle from going beyond the interval from –π to π or discontinuous. The proportional, integral, and derivative parameters will be reported in Table 2.

Table 2. The parameters of controllers in BalanceBot

|  |  |  |  |
| --- | --- | --- | --- |
| Controllers | Kp | Ki | Kd |
| Pitch Angle Controller |  | 20.0 | 0.15 |
| Linear Velocity Controller |  | 0.0 | 0.005 |
| Turning Velocity Controller | 7000.0 | 0.0 | 600.0 |
| Position Controller  Heading Controller | 1.0  0.00008 | 0.0  0.0 | 0.01  0.000001 |

## Odometry and Tracking

Odometry utilizes the data from the wheel encoders to estimate the pose of the Balancebot over time as shown in eq. (1) assuming no side slip of the Balancebot ().

(1)

Where are the location and heading of the Balancebot at time ; are the change of x coordinate and heading during one odometry update.

The trajectory of Balancebot between two odometry samplings can be approximated by an arc (Fig. 1). According to the odometry model (ref), and can be calculated from eq. (2).

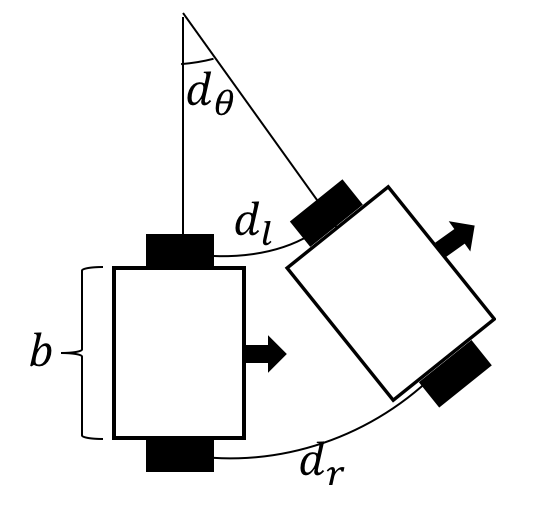


Fig 1: Balancebot Odometry Estimation

(2)

Where and are the distance traveled by the left and right wheels, and is the wheel separation distance. and can be estimated from the wheel encoder measurements as shown in eq.3.

u (3)

Where is the distance traveled by wheel, is the encoder resolution, u is the gear ratio, is the wheel diameter.

## 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 [2].

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**  **end for**  **end 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.

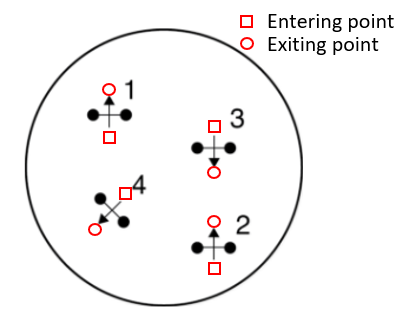


Fig. xx: An example of gate configuration. Square represent entering point and circle is exiting point.

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

(20)

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 and Discussion

## Controllers

### Impulse Response of Pitch Angle

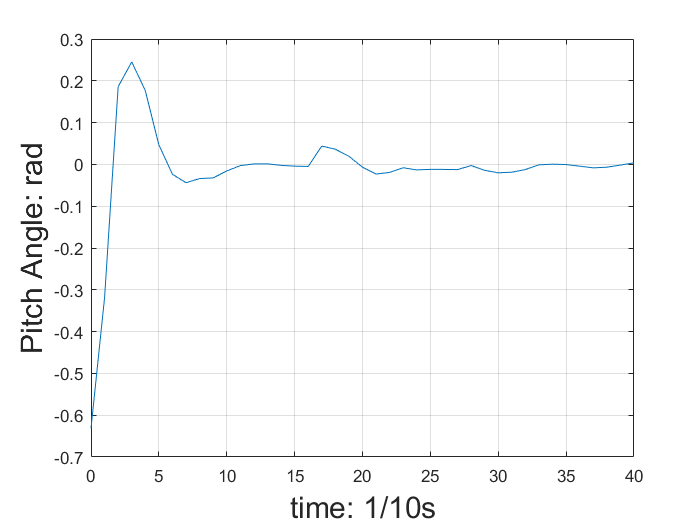


Fig. XX Impulse Response of Pitch Angle

As shown in Fig. XX, the impulse response of pitch angle has an overshoot towards the impulse signal. But the pitch angle goes back to equilibrium in about 1 second. Also, no other overshoot is generated. If we judge the response by the performance saw by our eyes, we find that the BalanceBot can go back to straight up position in a time of approximately 1 second with oscillation no more than once. Then it will keep at equilibrium position without going away from it.

### Impulse Response of Linear Velocity

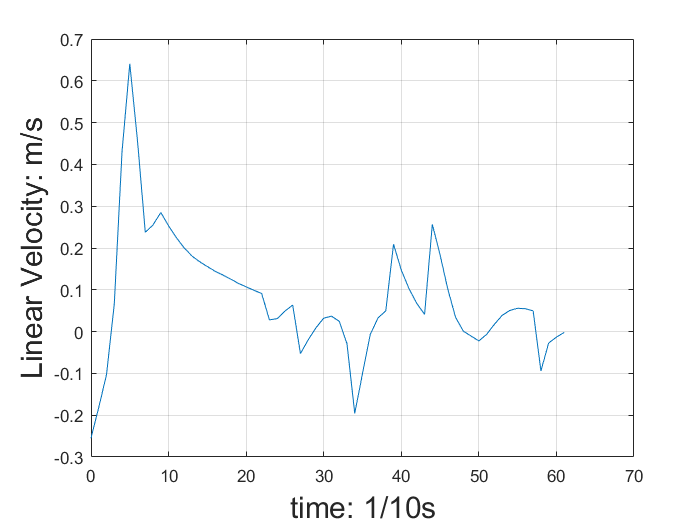


Fig. XX Impulse Response of Linear Velocity

As shown in Fig. XX, the impulse response of linear velocity has an overshoot larger than the amplitude of input impulse and it will oscillate around the equilibrium point more than once. The system will go back to stable in more than 6 seconds. Compared to the Pitch Angle Controller, the Linear Velocity Controller generates larger overshoot and larger settling time. According to the impulse response, the Linear Velocity needs to be tuned again.

### Impulse Response of Turning Velocity

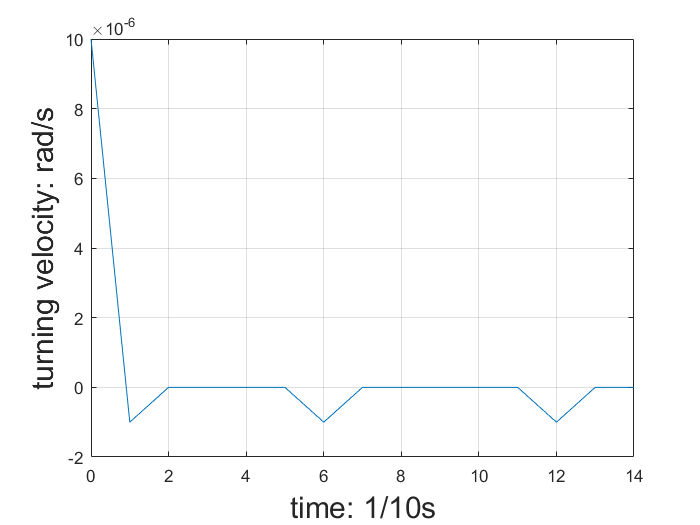


Fig. XX Impulse Response of Turning Velocity

As shown in Fig. XX, the impulse response of turning velocity has small overshoot which is just 10% of the amplitude of input signal. Also, its settling time is just 0.2 second. It means that the angular can be stabilized in just 0.2 second and will not go unstable. Among all three controllers, Turning Velocity Controller has the smallest overshoot and the smallest settling time.

## Odometry

To validate the result of dead-reckoning, the BalanceBot drives around a 1m by 1m square path four times. The trajectories determined by the odometry and the optitrack are shown in Fig. xx.

Add a discussion here

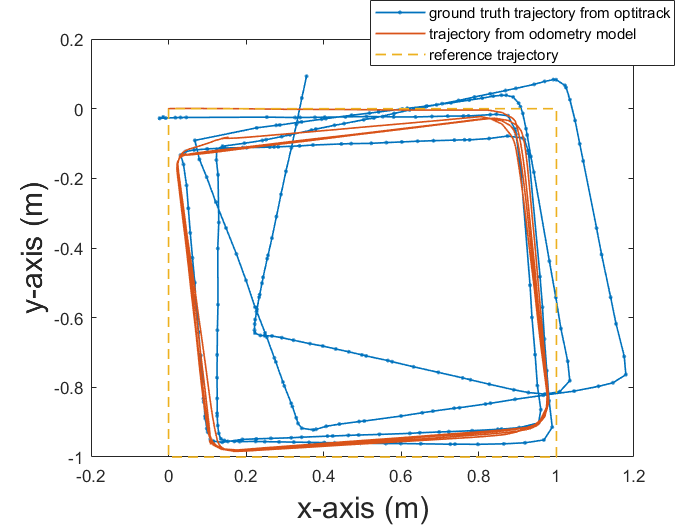


Fig. xx Comparison of the trajectory determined by odometry and optitrack

## Path planning

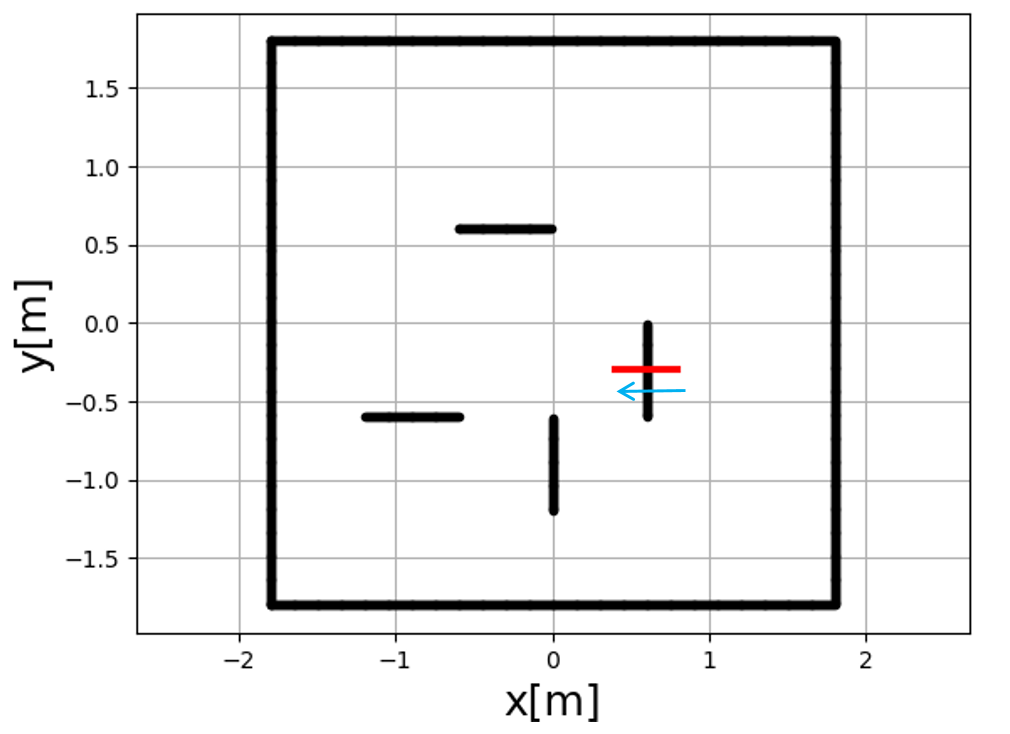
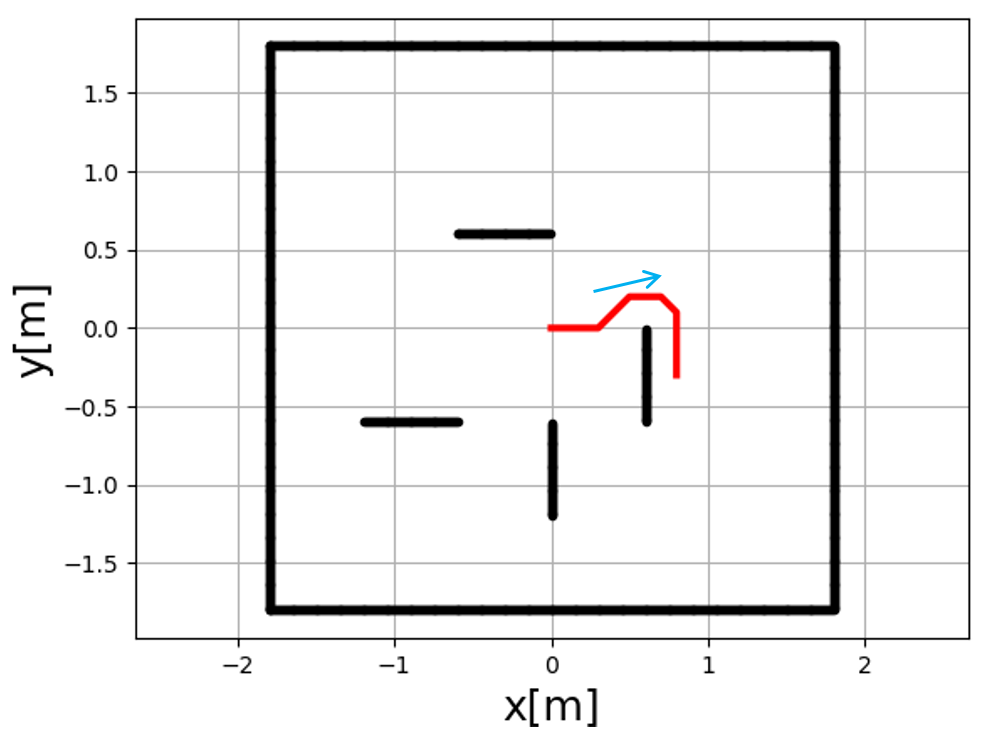
Based on the gate positions obtained from optitrack, we firstly run a Python simulation to generate the waypoints in the whole path. The gate positions are shown in following table.

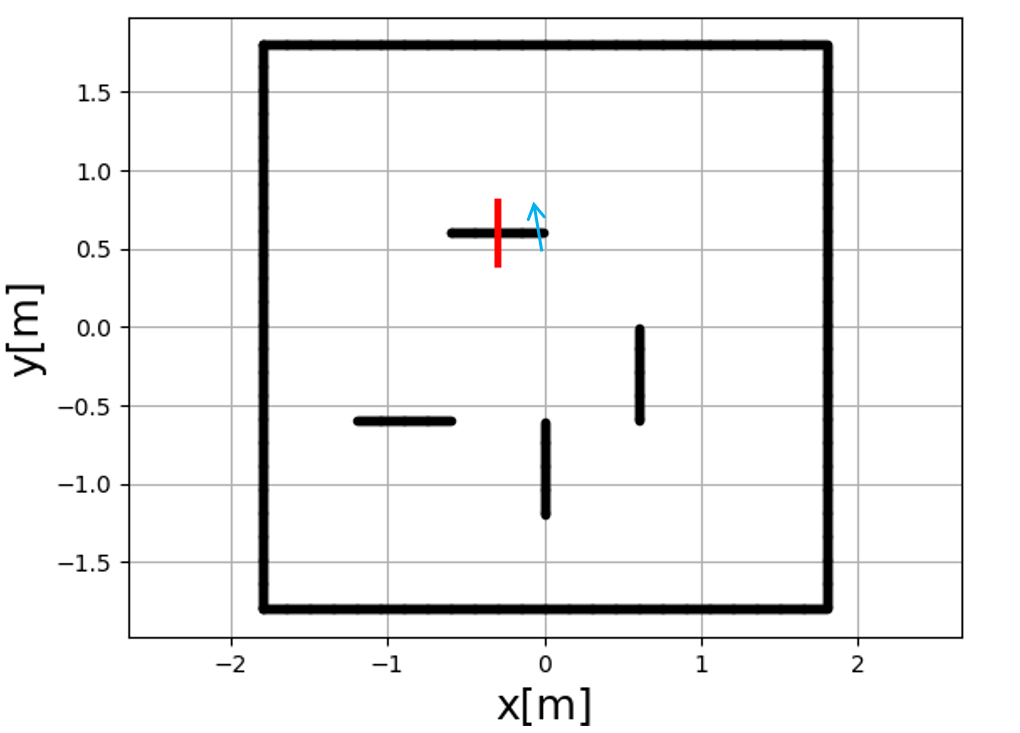
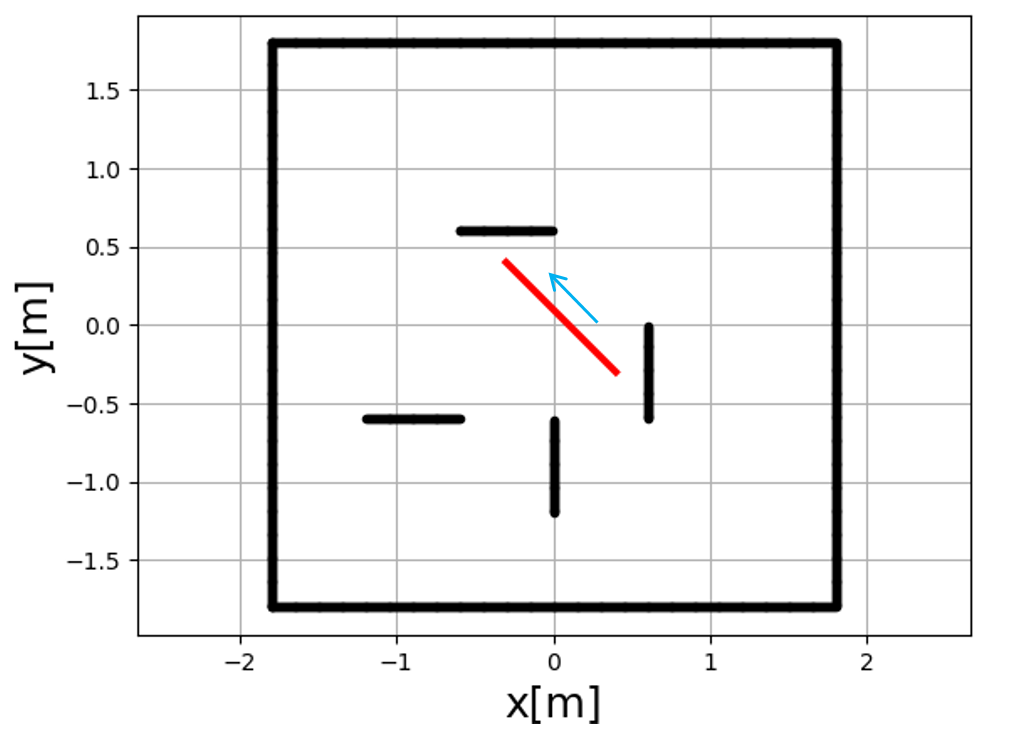
TABLE xx

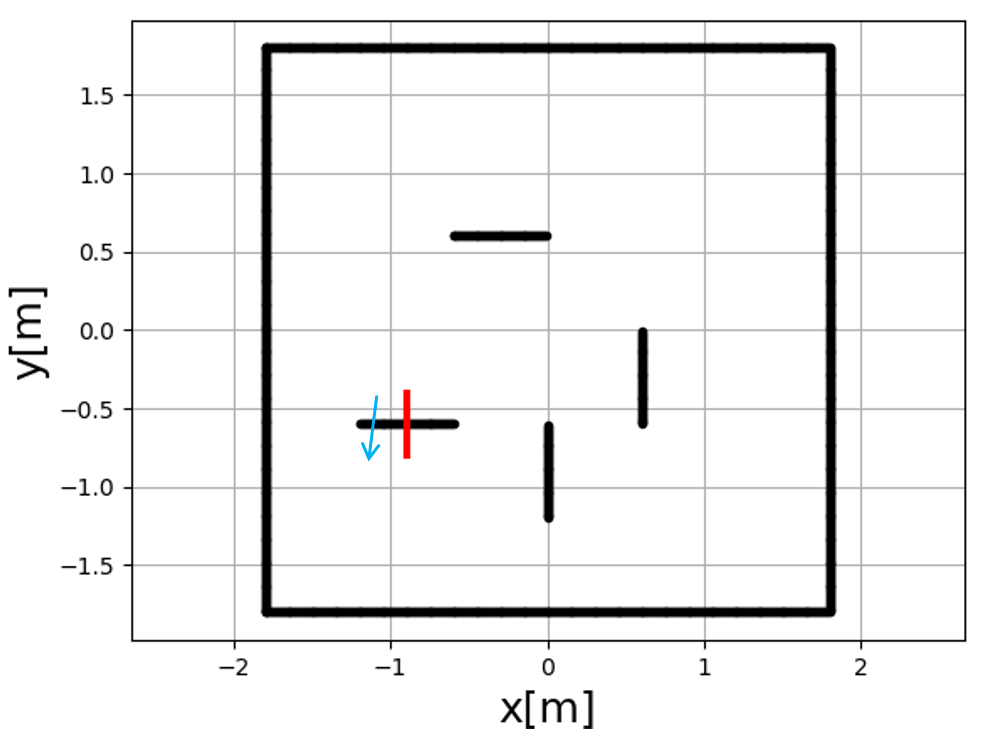
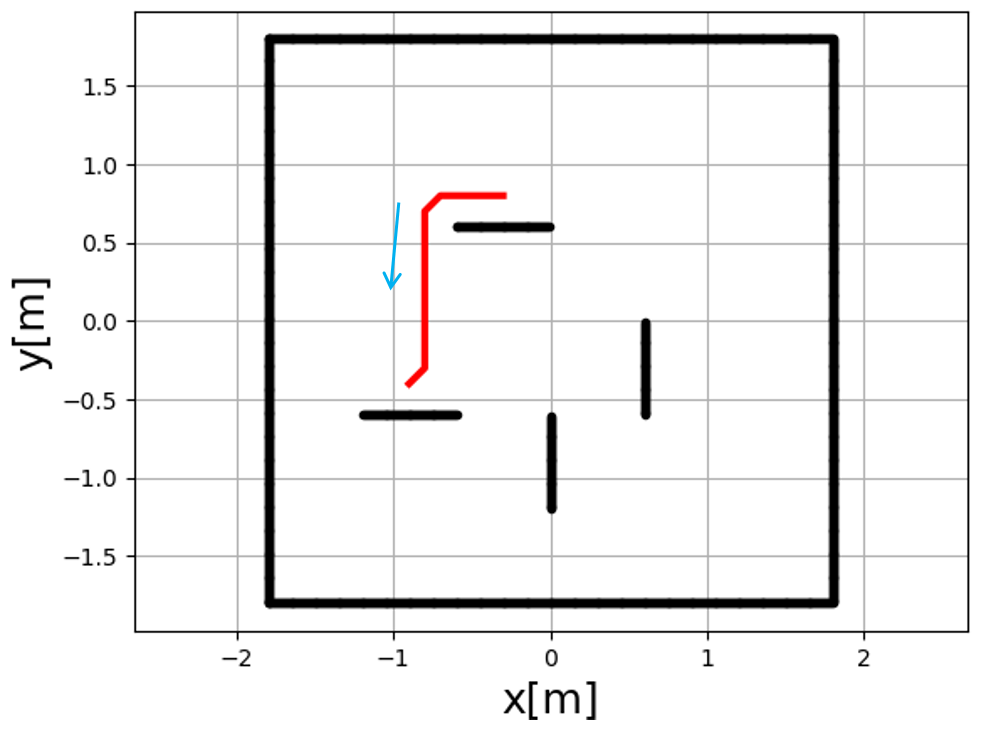
The position of each gates

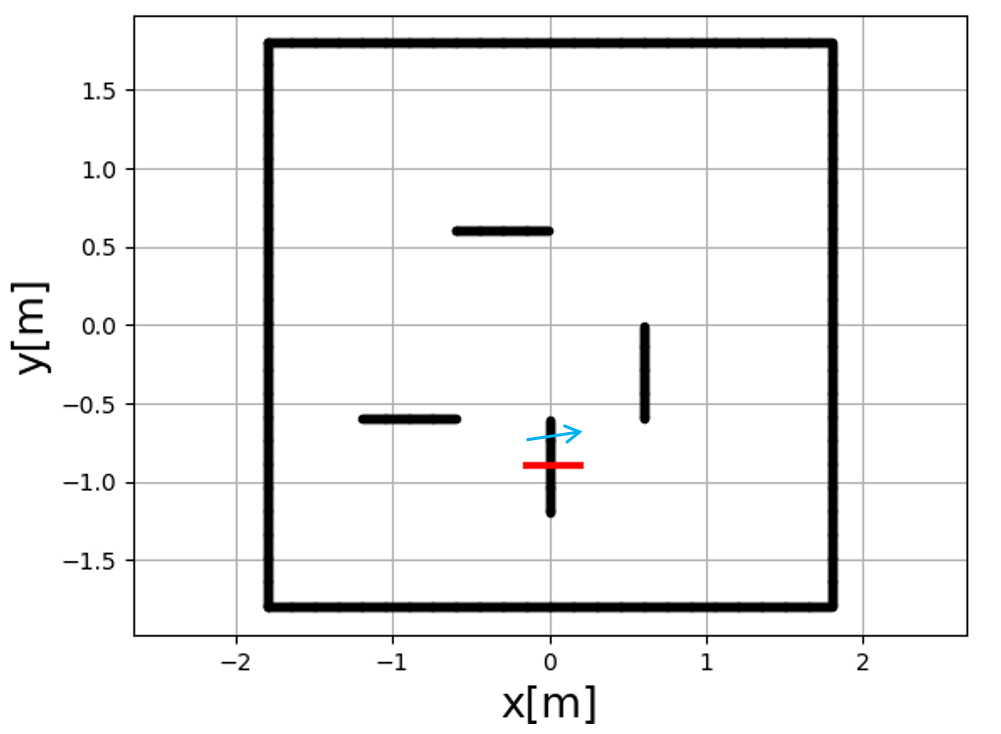
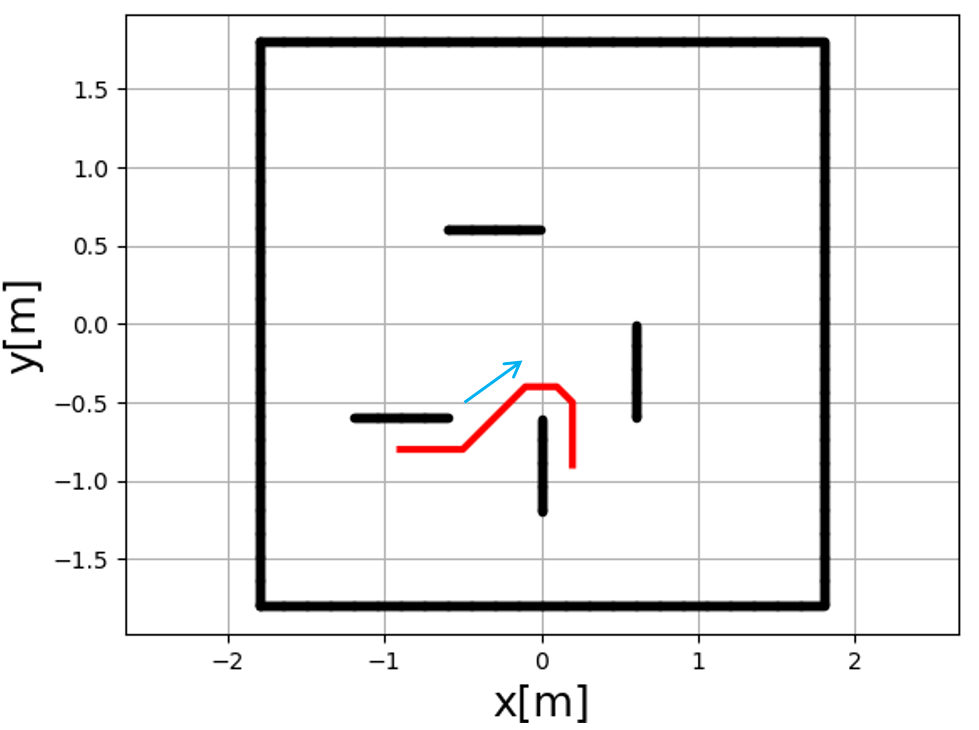
|  |  |  |
| --- | --- | --- |
| # of gate | Left [m] | Right [m] |
| 1 | (0.062, -0.064) | (0.062, -0.001) |
| 2 | (-0.061, 0.065) | (0.002, 0.062) |
| 3 | (-0.064, -0.061) | (-0.121, -0.063) |
| 4 | (0.001, -0.123) | (0.001, -0.061) |

The result of path is shown in Fig. xx.









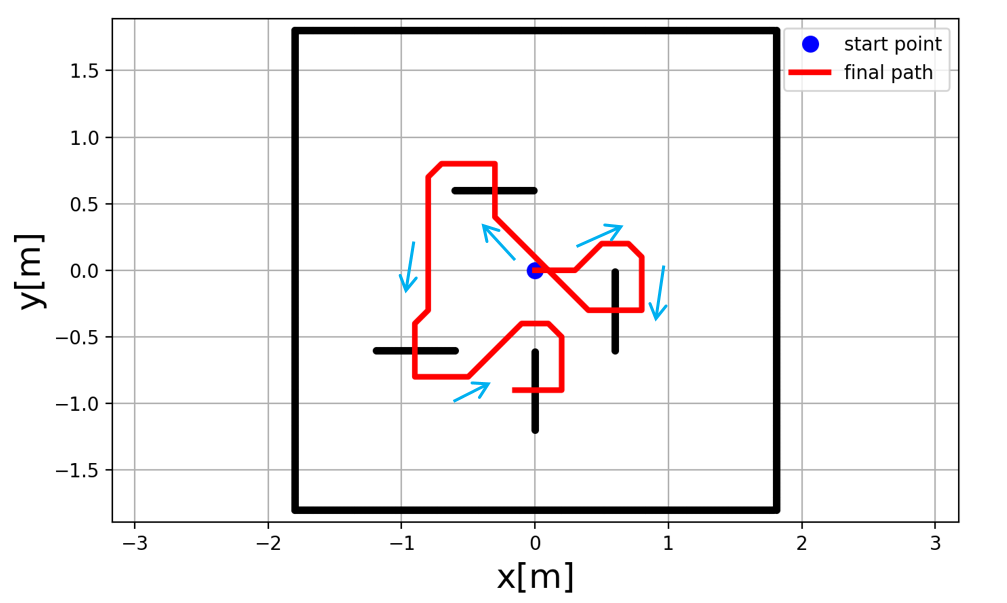


Fig. xx: 8 pieces of partial paths and the whole path

# 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

*A.1 The specifications of BalanceBot*

|  |  |
| --- | --- |
| Parts | Dimension |
| Motor Gear Ratio |  |
| Encoder Resolution  Wheel Diameter | 48 PPR  0.008006 m |
| Wheel Base  Maximum width | 0.21165 m  0.245 m |

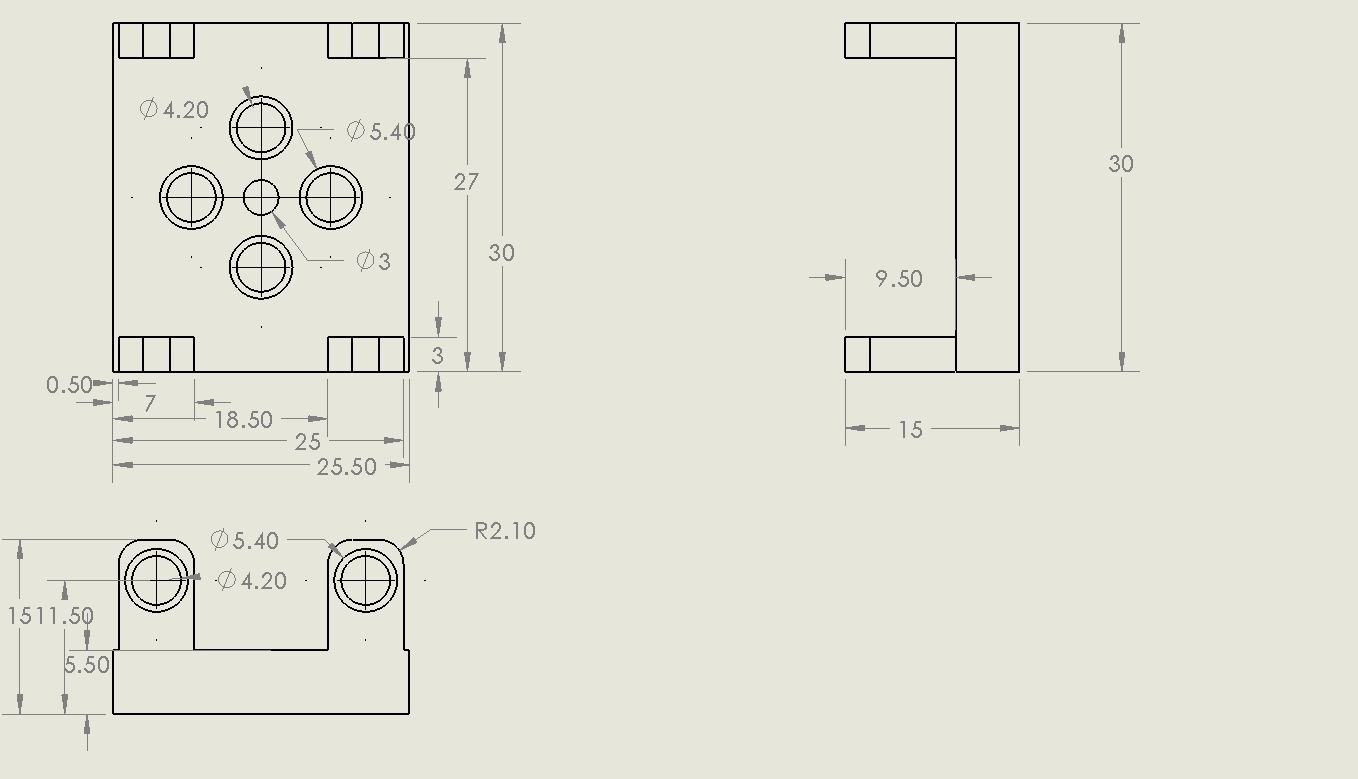
TABLE A.1

BOM

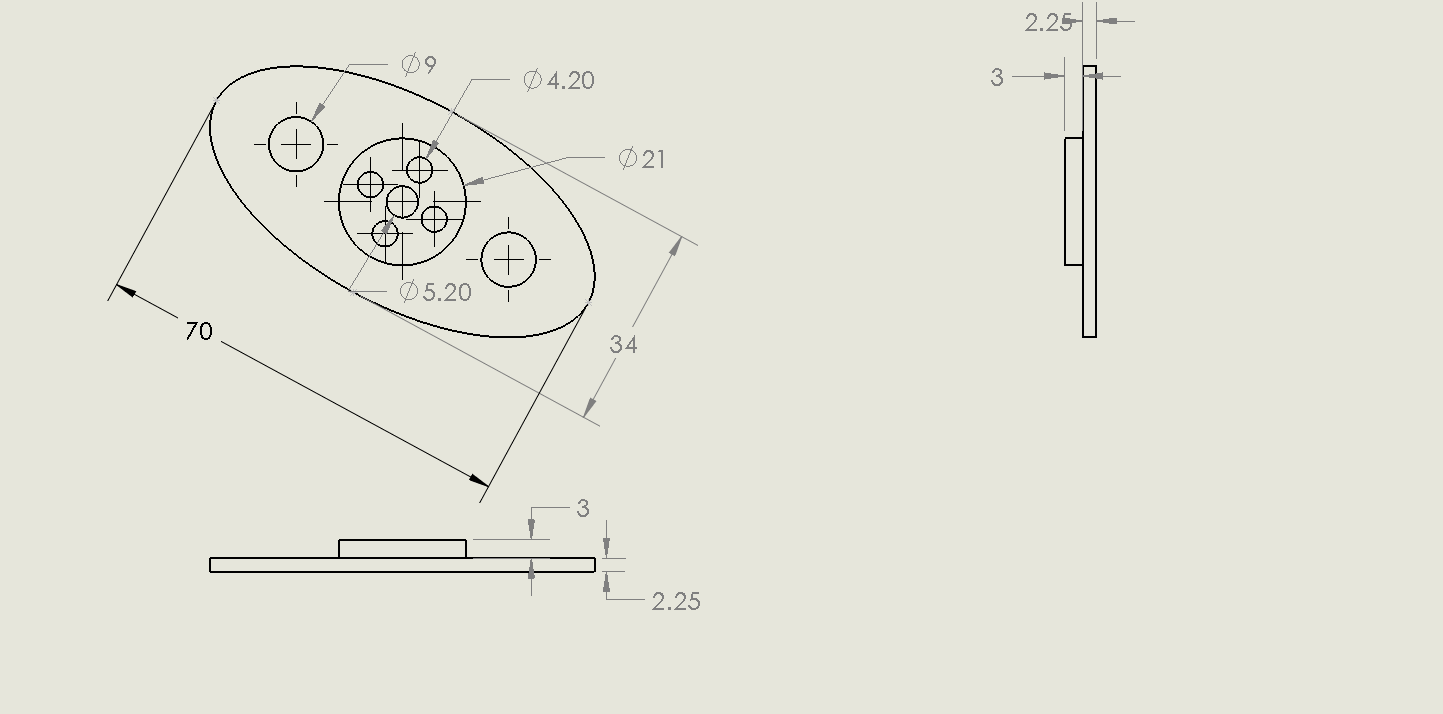
|  |  |
| --- | --- |
| Part name | # of parts |
| Beaglebone Green | 1 |
| Mobile Robotics Cape | 1 |
| 3 cell, 1500mAh Lithium Polymer battery | 1 |
| battery monitor | 1 |
| 20.4:1 Metal Gearmotor 25Dx50L mm MP 12V with 48 CPR Encoder | 2 |
| DRV8801 Single Brushed DC Motor Driver Carrier | 1 |
|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

# Appendix d: cad drawing

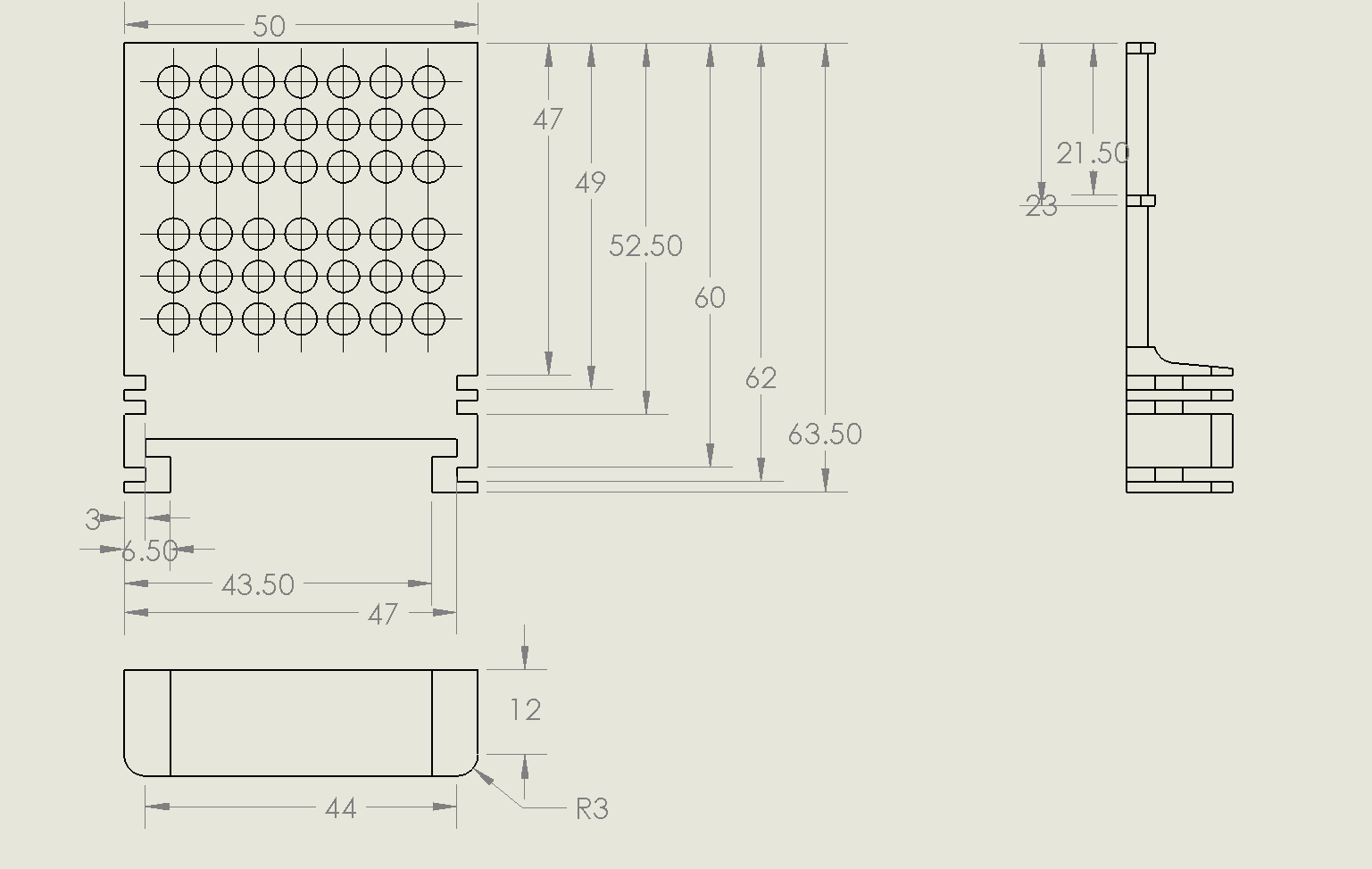
Connector



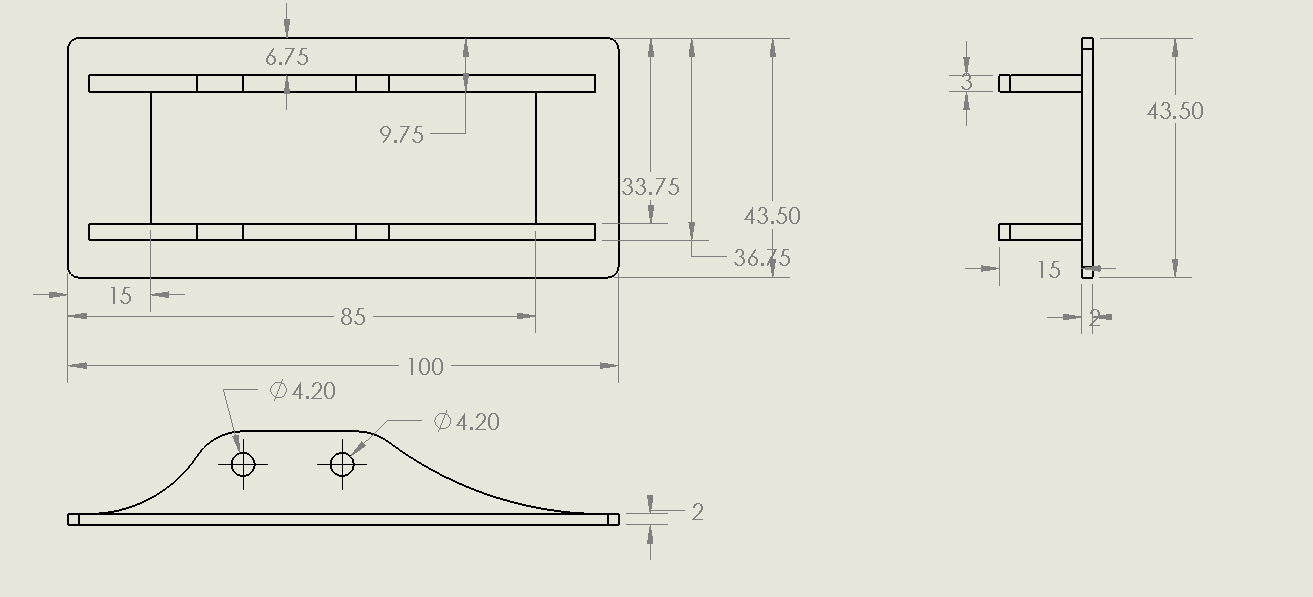
Ellipse with spacer



Gripper hand



Rail



1. [↑](#footnote-ref-2)