Two-Wheel Balanced Robot

Weilun Peng, Da Li, and Jia Shi

***Abstract* — We proposed a methodology of automating a two-wheeled self-balanced robot (the BalanceBot) to search and follow a path with a given destination in a constrained arena. We designed PID controllers to balance and drive the robot without falling over. 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 the shortest path to a target location. All these functions enables the robot to automatically find and follow a path in a maze with four gates.**

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

I. Introduction

In this lab, we are tasked to build a two-wheeled robot, which is capable of performing the following four tasks: 1) autonomously drive in a 1m by 1m square while always keep balancing using the odometry as feedback; 2) autonomously drive in a straight line for 11m using dead reckoning without falling over; 3) manually drive the robot to pass four gates in specified directions in a maze; 4) autonomously drive through the gates 3 times.

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

II. Methodology

A. Motor Model

The motor parameters are measured according to [1]. The motor parameters include (a) motor resistance r measured directly from the multimeter; (b) motor current i and voltage V measured directly from the multimeter (under 0.5, 0.7, and 1.0 PWM); (c) no load speed ω calculated using eq. 1 under 1.0 PWM; (d) motor constant K calculated using eq. 2; (e) stall torque τ calculated using eq. 3. Table 1 shows the summary of motor parameters.

$$\omega = \frac{2\pi \cdot enc}{R_{enc}\, u\, dt} \quad (1)$$

Where enc is the wheel encoder measurement; R_{enc} is the wheel encoder resolution; u is the gear ratio; dt is the time elapse between two wheel encoder measurements (in seconds).

$$K = \frac{V - i \cdot r}{\omega} \quad (2)$$

$$\tau = \frac{KV}{r} \quad (3)$$

Table 1. Motor Parameters

	Left Motor	Right Motor
Motor resistance R (ohm)	4.7	5.5
Motor constant K	0.283	0.278
No load speed w_{nl} (rad/s)	39.85	40.42
Stall torque τ $(N \cdot m)$	0.67	0.55

B. Controllers

Theoretically, the BalanceBot has an equilibrium pitch angle which points to upright, but this configuration results in an unstable system. In this case, any input or response to the system will push the BalanceBot away from the equilibrium pitch angle, however, the system is incapable to recover 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 staying at the equilibrium pitch angle with a desired linear and angular velocity. In addition, position controller and heading controller were implemented to drive the BalanceBot to the target location with a desired pose.

The block diagram of the cascade controller system is shown in Fig. 1. The most inner loop has two parallel controllers: *Pitch Angle Controller* and *Turning Velocity Controller*. The output of these two controllers controls the motion of the BalanceBot. The outer loop of these two controllers are *Heading Controller* and *Linear Velocity Controller*. The *Heading Controller* feeds the desired turning velocity into the *Turning Velocity Controller* and the *Linear Velocity Controller* provides the desired pitch angle to the *Pitch Angle Controller*. The outermost controller is the *Position Controller* that can generate the desired linear velocity for the *Linear Velocity Controller*. All five controllers in this diagram are PID controllers. We manually tuned the PID controllers following the steps described as follows. First, we increased Kp till the oscillation happened. Then, we gradually decreased Kp to a value such that oscillation just disappears. Second, we started tuning Kd by first setting Kd to the same value as Kp. Then we noticed that there were oscillations and the BalanceBot started to tremble. As a result, we gradually decreased Kd until removing these phenomena. To remove the interference of noises in the signal, the derivative terms in all five controllers are filtered by a first order low-pass filter with a cut-off frequency of 40 Hz. Lastly, Ki is added if the BalanceBot always drifted to one direction. By adjusting Ki, this drift could be reduced. In our system, Ki is only necessary for the *Pitch Angle Controller* to reduce the error.

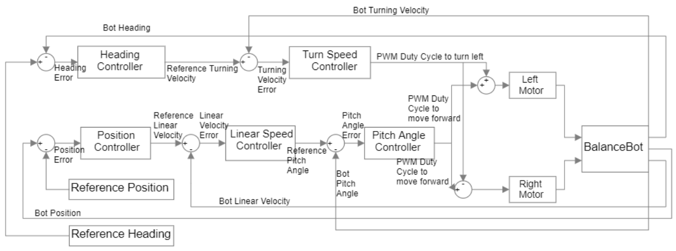


Fig. 1. Block diagram of the cascade controller system

As shown in Fig. 2, the pitch angle is defined as the angle between the vertical symmetry axis of the BalanceBot and the upright. The heading angle is defined as the angle between horizontal symmetry axis of the BalanceBot and the initial facing direction.

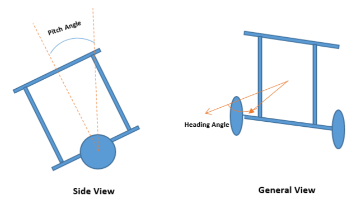


Fig. 2. Simplified Model of BalanceBot

1) Pitch Angle Controller: The *Pitch Angle Controller* takes the target pitch angle and the pitch angle measured from IMU. Since the pitch angle from IMU has a nonzero value when the BalanceBot is initialized at the upright state. We wait 15 seconds before running the BalanceBot on the arena to calibrate the IMU. The mean value of the first three pitch angle measurements from the IMU is adopted as the pitch angle offset of the current operation. It should be noted that the raw pitch angle measurement is not continuous when balancing around the upright state. Hence, we subtracted 2π from the pitch angle measured by the IMU if it is greater than 0 such that the pitch angle is symmetric with respect to 0 and changes continuously when the BalanceBot balances forwards and backwards. The *Pitch Angle Controller* takes the angle difference between the targeted pitch angle and the actual pitch angle measured from IMU as input and outputs a PWM to keep the BalanceBot at a balanced state during transition, rotation or idling. The corresponding Kp, Ki, Kd are reported in Table 2.

2) Linear Velocity Controller: The *Linear Velocity Controller* takes the target linear velocity and the actual velocity of the BalanceBot measured from the wheel encoder. The actual linear velocity is calculated from the encoders of left and right motors. Before each update, the encoders are reset to 0 such that the encoder measurements represent the changes during that period. The rotational velocity ω of each wheel (in RPM) is calculated from eq. 4:

$$\omega = \frac{enc\ \times f \times 60}{u \times R_{enc}} \quad (4)$$

Where enc is the wheel encoder measurement between two updates; R_{enc} is the wheel encoder resolution; u is the gear ratio; f is the sampling rate.

The actual linear velocity v (in m/s) is calculated using Eq. 5.

$$v = \frac{\omega_l + \omega_r}{2 \times 60} \times \pi \times D \quad (5)$$

Where ω_l, ω_r are the rotational velocity of the left and right wheel; D is the wheel diameter.

The *Linear Velocity Controller* takes the difference between the desired linear velocity and the actual linear velocity and outputs a desired pitch angle. The corresponding Kp, Ki, Kd are reported in Table 2.

3) Turning Velocity Controller: the *Turning Velocity Controller* follows the same logic of implementation as the *Linear Velocity Controller*. It takes the reference turning velocity and the actual angular velocity that the BalanceBot was turning to the left. The speed of left and right motors is calculated using the eq. 4 discussed above. The actual turning speed (in rad/s) is calculated using Eq. 6:

$$\frac{\omega_r - \omega_l}{b \times 60} \times 2\pi \times D \quad (6)$$

Where b is the wheel separation distance.

The *Turning Velocity Controller* takes the difference between the desired turning velocity and the actual turning velocity and outputs a PWM duty cycle which is the fraction of one period of a PWM signal. The duty cycle controls the difference of speed between the left and right wheels, which can turn the BalanceBot to a desired direction with a desired angular velocity. If the output is positive, the BalanceBot makes a left turn. Otherwise, the BalanceBot makes a right turn. The corresponding Kp, Ki, Kd are reported in Table 2.

4) Position Controller: The *Position Controller* enables the BalanceBot to stay or move to a desired location. The *Position Controller* takes the distance between the current position and the desired position as an input and outputs a desired linear velocity (in m/s). The input distance d (in m) was calculated from eq. 7:

$$d = \sqrt{(x_t - x_c)^2 - (y_t - y_c)^2} \quad (7)$$

Where x_t, y_t are the coordinates of the desired location; x_c, y_c are the coordinates of the current location. The corresponding Kp, Ki, Kd are reported in Table 2.

5) Heading Controller: The *Heading Controller* takes the difference between the desired heading and the actual heading to enable BalanceBot turning to a desired heading angle. The actual heading angle is calculated from the odometry model. The desired heading angle (θ) is calculated from eq. 8:

$$\theta = \tan^{-1}(\frac{y_t - y_c}{x_t - x_c}) \quad (8)$$

The difference between the desired and current heading angle (denoted as $\Delta\theta$) can be outside of the range $[-\pi, \pi)$. We clamp $\Delta\theta$ to $[-\pi, \pi)$ and covert it to a continuous variable using eq. 9:

$$\Delta\theta = \begin{cases} -\Delta\theta + \pi, & \text{if } \Delta\theta > \pi \\ -\Delta\theta - \pi, & \text{if } \Delta\theta < -\pi \end{cases} \quad (9)$$

The actual heading angle is also processed using eq. 9. After this process, the clamped actual heading angles is subtracted from the difference of heading angle, which prevents the difference of heading angle from going beyond the interval from –π to π. The corresponding Kp, Ki, Kd are reported in Table 2.

Table 2. Parameters of PID Controllers of the BalanceBot

Controllers	Kp	Ki	Kd
Pitch Angle Controller	3.0	20.0	0.15
Linear Velocity Controller	0.1	0.0	5.0×10^{-3}
Turning Velocity Controller	7.0×10^{3}	0.0	6.0×10^{2}
Position Controller	1.0	0.0	0.01
Heading Controller	8.0×10^{-5}	0.0	1.0×10^{-6}

C. Odometry and Tracking

Odometry utilizes data from the wheel encoders to estimate the pose of the Balancebot (x, y, θ) over time as shown in eq. (10) assuming no side slip of the Balancebot ($\Delta y = 0$).

$$\begin{bmatrix} x_{t'} \\ y_{t'} \\ \theta_{t'} \end{bmatrix} = \begin{bmatrix} x_t + \Delta x \cos\theta \\ y_t + \Delta x \sin\theta \\ \theta_t + \Delta\theta \end{bmatrix} \qquad (10)$$

Where $[x_t, y_t, \theta_t]^T$ is the location and heading of the Balancebot at time t; $\Delta x, \Delta\theta$ are the changes in x direction and heading during one odometry update.

The trajectory of the Balancebot between two odometry updates can be approximated by an arc (Fig. 3). According to the odometry model [2], Δx and $\Delta\theta$ can be calculated from eq. 11, respectively.

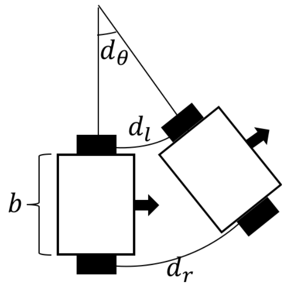


Fig. 3. Balancebot Odometry Estimation

$$\Delta x = \frac{d_l + d_r}{2}, \ \Delta\theta = \frac{d_r - d_l}{b} \qquad (11)$$

Where d_l and d_r are the distances traveled by the left and right wheels, and b is the wheel separation distance. d_l and d_r can be estimated from the wheel encoder measurements from eq. 12.

$$d = R_{enc}\, u\, \pi D \qquad (12)$$

Where d is the distance traveled by a wheel, R_{enc} is the encoder resolution, u is the gear ratio, D is the wheel diameter. Table 3 presents all parameters in the odometry model.

Table 3. Parameters of Odometry Model

Parameter	Value
R_{enc}	48
u	20.4
D (m)	80.06×10^{-3}
b (m)	211.65×10^{-3}

D. Path Planning

In this section, we developed a planner to move the BalanceBot to a desired location without collision. The planner selects waypoints along the trajectory generated from the A* search algorithm. The waypoints are also decided by using the real-time coordinates from odometry and Optitrack.

1) Description of path planning algorithm: In order to complete task 4, we choose A* search algorithm to find the shortest path to a desired location. Comparing with Dijkstra's algorithm, A* search algorithm is much faster due to the merit of best-first search.

In this task, we simplify our BalanceBot as a single point and all obstacles (i.e., gates) as circles with a certain radius. If the BalanceBot drives into the circular region, a collision will happen. To this end, we set the radius of obstacles as one half of the maximum width of the BalanceBot, which is 0.245 m.

With the configuration space of the world, the A* search algorithm constructs a tree of paths originated from the starting node and expands paths by one step each time until one of the possible paths reaches the predetermined goal node. At each iteration of its main loop, A* algorithm selects the path with the minimum total path cost, which is presented in eq. 13.

$$f(n) = g(n) + h(n) \qquad (13)$$

Where n is the last node/point on the path, $g(n)$ is the cost of the path from the starting node to node n; $h(n)$ is a heuristic function that estimates the cost from node n to the goal; $f(n)$ is the estimated total cost of path through node n to the goal [3].

In this task, we define $h(n)$ as the Euclidean distance from the location of the last reached node n to the goal, which can be calculated from eq. 14. This heuristic is reasonable since it does not overestimate the cost to reach the goal.

$$h(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2} \qquad (14)$$

Where (x_n, y_n) and (x_{goal}, y_{goal}) are the coordinates of current and goal node, respectively.

With this heuristic, we can calculate the estimated total cost of each possible path and use priority queue to repetitively add nodes with the minimum estimated cost to the current path. During each iteration, the node with the lowest $f(n)$ is removed from the queue, and the f and g values of its adjacent neighbors are updated accordingly with additional adjacent distance and then the neighbors without collision are added to the queue. The algorithm continues until a goal node has a smaller f value than any nodes in the queue or the queue is empty. In this case, the f value of the goal is the length of the shortest path, since h value at the goal is zero in the heuristic. 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

```
$n_{start}: dis_{start} = 0, parent_{start} = none, visited_{start} = false$
$visit_{queue} = n_{start}$
while ($visit_{queue}$ != $empty$) && $n_{current}$ != $n_{goal}$
  dequeue: $n_{current} = node\ in\ visit_{queue}\ with \min f$
  $visited_{current} = true$
  for each nbr in not visited(adjacent($n_{current}$))
    if !collision(nbr)
      enqueue: nbr to $visit_{queue}$
      if $g(nbr) > g(n_{current}) + dist(nbr, n_{current})$
        $parent_{nbr} = n_{current}$
        $g(nbr) = g(n_{current}) + dist(nbr, n_{current})$
```

```
            $f(nbr) = g(nbr) + dist(nbr, n_{goal})$
        end if
    end for
end while
```

After finding waypoints from the starting point to the goal, we calculate the gradients of the adjacent waypoints. We set a threshold of 0.5 rad (approximately 30 degrees). If the gradient is greater than 0.5, it means a turning is required between these two waypoints. Otherwise, the waypoints are removed from the planned trajectory.

2) Gate waypoint placing: The location of the left and right gates, as well as, the direction to pass each gate are shown in Fig. 4. For each gate, we select two waypoints (one on each side of the gate) to represent the entering point and exiting point.

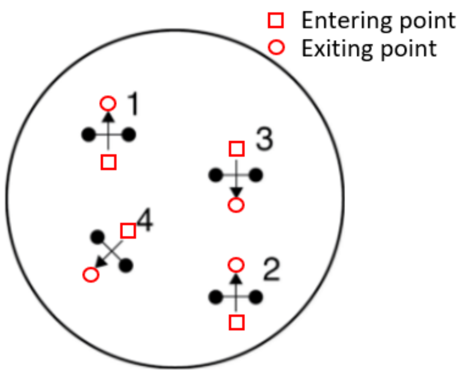


Fig. 4. Configuration of Gates for Task 4

The entering and exiting waypoints are calculated from the coordinates of left and right gates using eq. 15:

$$
\begin{aligned}
x_{gatecenter} &= \frac{x_{gateleft} + x_{gateright}}{2} \\
y_{gatecenter} &= \frac{y_{gateleft} + y_{gateright}}{2} \\
v_x &= y_{gateleft} - y_{gateright} \\
v_y &= x_{gateright} - x_{gateleft} \\
x_{enter} &= x_{gatecenter} - \frac{w_{max} v_x}{2} \\
y_{enter} &= y_{gatecenter} - \frac{w_{max} v_y}{2} \\
x_{exit} &= x_{gatecenter} + \frac{w_{max} v_x}{2} \\
y_{exit} &= y_{gatecenter} + \frac{w_{max} v_x}{2}
\end{aligned} \tag{15}
$$

Where w_{max} is the maximum width of the robot; (x_{enter}, y_{enter}) and (x_{exit}, y_{exit}) are entering and exiting waypoints we select for each gate.

3) Final implementation: The robot moves from the entering point to the exiting point to pass each gate. Once the robot reaches the exiting point, the gate is regarded as open and will not be considered as an obstacle in the remaining path-finding task. In this case, A* search algorithm is implemented again to find the subsequent path which starts from the last exiting point to the next entering point.

The competition consists of a total of 4 gates. Thus, 4 entering points and 4 exiting points are generated in addition to the starting point and goal point. Between two adjacent waypoints, an RTR planner is executed to drive the BalanceBot in a straight line. The A* search algorithm is implemented in Python to generate waypoints data for the competition and the resulting path is presented and discussed in the result section.

4) *RTR planner*: Given a set of waypoints generated from the A* search algorithm, the RTR planner traverses each waypoint in the predetermined order by first rotating to face the target waypoint, then translating towards it, and finally rotating to face the next target waypoint. As the waypoints generated from the A* search algorithm are not within the proximity of obstacles, the BalanceBot should be able to follow the defined trajectory without colliding the obstacles.

III. Results and Discussion

A. Controllers

1) Impulse Response of Pitch Angle

As shown in Fig. 5, the impulse response of pitch angle has an overshoot towards the impulse signal. However, the pitch angle goes back to the equilibrium state in about 1 second. Other than that, no additional overshoots in pitch angle are generated. By visually observing the performance of our BalanceBot, we found that the BalanceBot can recover to the upright state in approximately 1 second with only one oscillation and then remain in the balanced state.

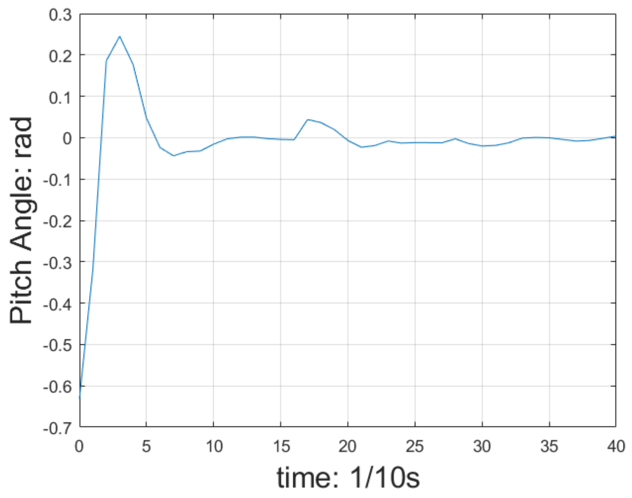


Fig. 5 Impulse Response of Pitch Angle

2) Impulse Response of Linear Velocity

As shown in Fig. 6, the impulse response of linear velocity has an overshoot which is larger than the amplitude of input impulse. The impulse response oscillates around the equilibrium state more than once. In this case, it takes more than 6 seconds for the BalanceBot to recover the equilibrium state. Compared to the *Pitch Angle Controller*, the Linear *Velocity Controller* generates a larger overshoot, as well as, a longer settling time. According to the impulse response, further tuning is required for the *Linear Velocity Controller*.

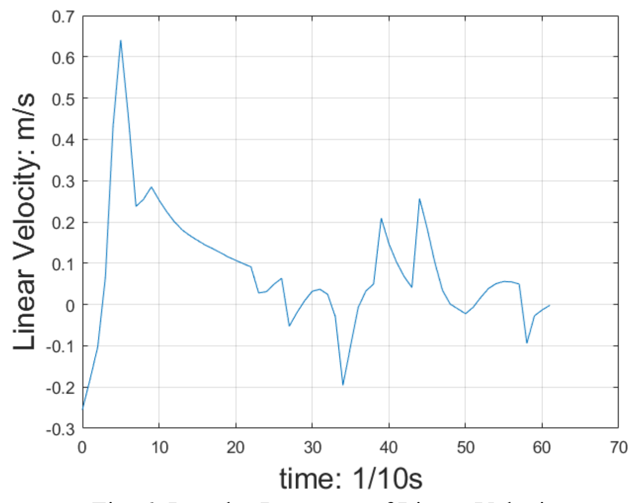


Fig. 6. Impulse Response of Linear Velocity

3) *Impulse Response of Turning Velocity*

As shown in Fig. 7, the impulse response of turning velocity has a small overshoot which equals to 10% of the amplitude of input signal and the settling time is only 0.2 second. It means that the angular speed can be stabilized in 0.2 second and remain in the equilibrium state afterwards. Among all three controllers, *Turning Velocity Controller* has the least overshoot and the shortest settling time.

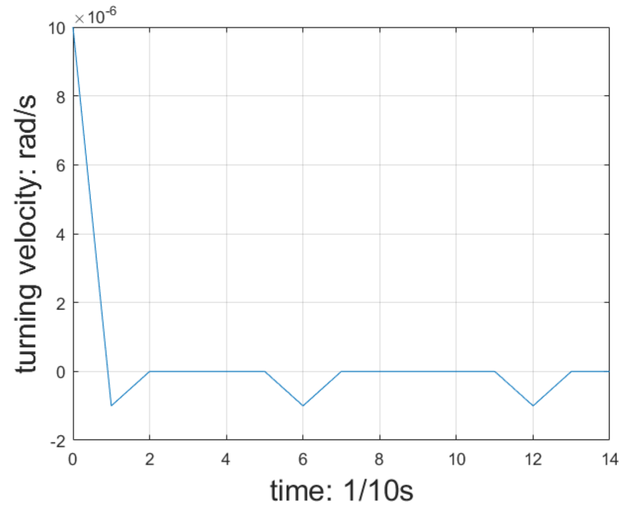


Fig. 7. Impulse Response of Turning Velocity

B. *Odometry*

To validate the results of dead-reckoning, the BalanceBot autonomously drives around a 1m by 1m square four times using the RTR planner. The waypoints are recorded to plot the trajectories determined by the odometry and the Optitrack (Fig. 8). Table 4 shows the mean differences between the Optitrack and odometry data (Optitrack - odometry).

Table 4. Comparison of the Odometry and Optitrack Data

Parameter	Mean Differences
x (m)	-0.0443
y (m)	-0.0346
θ (rad)	0.135

As shown in Fig. 8, the ground truth trajectory from Optitrack shows that the pose error is accumulated over time while the trajectory generated from the odometry model maintains a square shape. This is probably caused by the slippery of wheels when turning 90 degrees at each vertex.

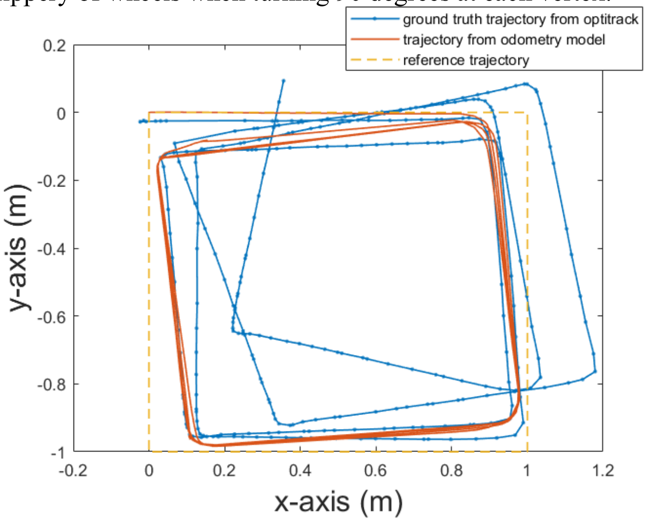


Fig. 8. Trajectory Determined by Odometry and Optitrack

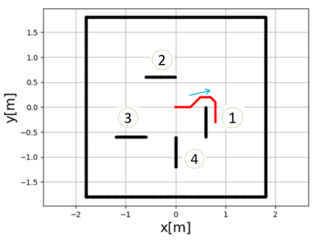
C. *Path planning*

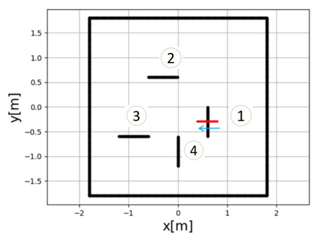
Based on the gate positions obtained from Optitrack, we firstly run a Python simulation to generate the waypoints in the full path. The coordinates of each gate are shown in Table 5.

Table 5. The Coordinates of Each Gate

# 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 trajectories generated by the A* search algorithm are shown in Fig. 9. The BalanceBot passes each gate in a predetermined order as specified in [4]. Fig. 10 shows the full trajectory to pass 4 gates in a predetermined order and direction.





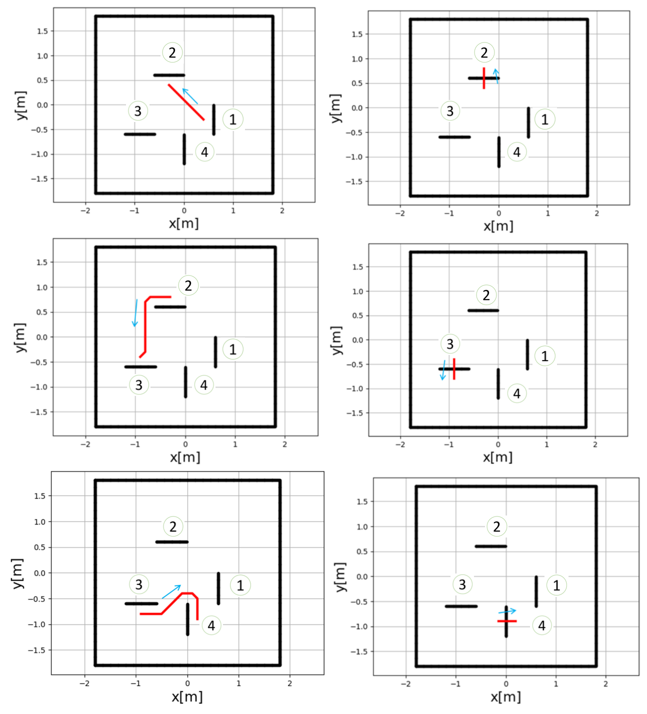


Fig. 9. Trajectories Generated by the A* Algorithm (gates are represented in solid black lines with the corresponding gate numbers; the red lines represent the trajectory of each segment, and the blue arrows represent the driving direction)

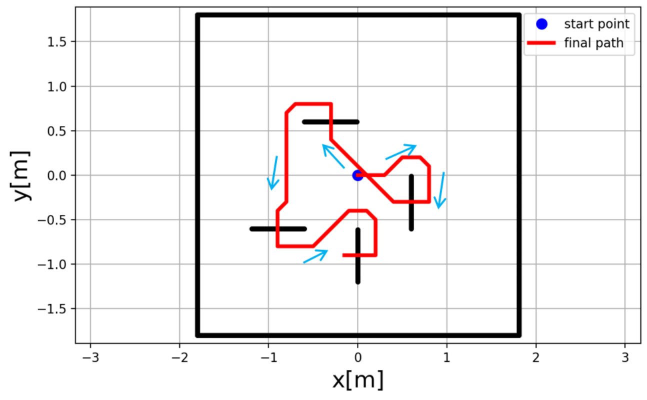


Fig. 10. The Full Trajectory Combining 8 Segments

IV. Conclusions

In this lab, a two-wheeled self-balanced robot is built to perform four tasks including 1) autonomously drive in a 1m by 1m square while always keep balancing using the odometry as feedback; 2) autonomously drive in a straight line for 11m using dead reckoning without falling over; 3) manually drive the robot to pass four gates in specified directions in a maze; 4) autonomously drive through the gates 3 times.

However, due to the hardware failure of our robot on the competition day, we were unable to perform the tasks. But we managed to finish all the checkpoints and successfully finished the required tasks offline.

For the PID controllers, we spent a lot of time to find the parameters to balance the robot at static and moving state. However, the performance is less than satisfactory as the BalanceBot still has a large oscillation. In the future we will use potentiometer to continuously change the input voltage to find the optimum parameters.

V. Reference

[1] Modeling a simple DC motor, P. Gaskell, 2018
[2] Odometry notes. Retrieved from https://drive.google.com/drive/folders/1lmedyQ4R7tQjH0Sl1lk0HPOicgNtW5SJ
[3] Lerner, Jürgen, Dorothea Wagner, and Katharina Zweig, eds. Algorithmics of large and complex networks: design, analysis, and simulation. Vol. 5515. Springer, 2009.
[4] BalanceBot project description. Retrieved from: https://docs.google.com/document/d/1YWWngsu0hmUbHfWZVZWMxbP-ww4bcN4qkOQ1ep20S20/edit#heading=h.lpkpuykbj583

Appendix A

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