12%/10% Motor Control
31%/30% Balance Control
12%/10% Turn Control
16%/20% Odometry -- odometry validation data wasn't used in any way to improve your odometry; odometry errors were only poorly discussed.
30%/30% Path planning
TOTAL: 100% IMPRESSIVE!!

***Abstract*—We propose a methodology of automating a two-wheel balanced robots, to plan and follow a path to a given destination in the constrained working arena while avoiding the obstacles in the way. We first design the controller for moving the robots in the working space without falling. Then we design the odometry and tracking algorithm for robot to track a given reference trajectory. Finally, the path planning algorithm enables the robot to plan a obstacle avoiding path. All these functions help the robot to go through a maize with open and closed doors automatically.**

Keywords-PID control, cascade control, navigation function.

I. Introduction

IN this report we will describe and discuss the implementation of a two-wheel self-balance robot using cascade PID control and path-planning function.

Section II shows the detailed design and implementation. The controllers take feedback from the 48 CPR encoder on DC motor and MPU9250 IMU which can measure the linear acceleration and angular velocity along 3 axes. They control the robot to stay balanced and move. The odometry, waypoint tracking mechanism and path planer make the robot navigate itself through obstacles.

Section III discusses the results. We plot the step responses on each controller, and discuss their sensitivity. We validate the odometry model, perform the accuracy and uncertainty test. Finally, we evaluate the path-planning function using the data from Optitrack.

II. Methodology

A. Controller

The balancebot itself is an unstable system and it's model is shown in Figure 1. In theory, the balancebot has an equilibrium position in vertical direction. But this equilibrium position is unstable, meaning any disturbance will drive the balancebot away from equilibrium position and the balancebot will never come back. Hence, we designed a cascade controller to control the balancebots' pitch angle, heading direction, velocity, and position.

As shown in the Figure 2, the cascade control includes four layers of loops. From the outside loop to inside loop, the controllers' priority increases. The proportional, integral, and derivative terms (Kp, Ki, Kd) of each layer's controller are shown in the appendix table III.

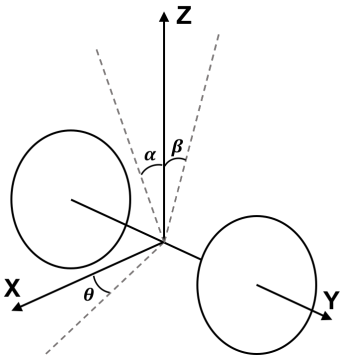

Fig. 1: Simplified model for balancebot.

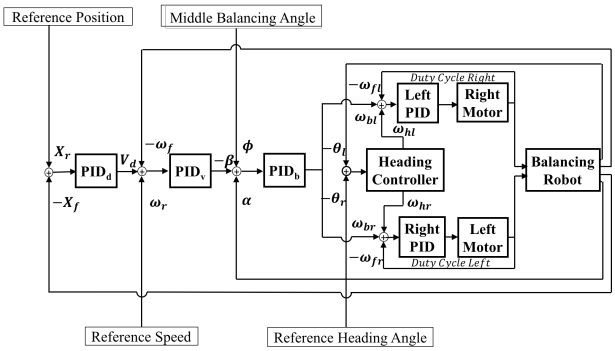

Fig. 2: The cascade controller for balancebot.

1) Left and right motor controllers: The innermost controllers are the left and right motor speed controllers which are with the highest priority. The input data of motor speed controller is the encoder data and the output is the dutycycle to DC motors. The input encoder data are also used as the input of velocity controller and odometry. Since the derivative term of PID control is sensitive to the signal noise, a median filter which takes the median of 5 adjacent data points is applied to filter out the sudden change of encoder signal. The motor speed controller is PI control. The derivative term is zero since the damping caused by the shaft's friction is already large enough to damp out the overshoot.

2) Heading and balance controllers: Outside the motor speed control are heading (PID_h) and balance controllers (PID_b). These two controllers are paralleled. The heading controller makes sure the balancebot's heading angle always meets the desired direction. The actual

heading direction is given by the odometry through task 1 to task 4. Once heading angle error occurs, the heading controller creates speed difference between left and right motors so that balancebot can make a turn. The heading controller only has a proportional term (Kp_h). There is no derivative term (Kd_h) and integral term (Ki_h) because the rolling friction is small enough and does not cause observable steady state error. Also, it is large enough to damp out the oscillation.

The balance controller makes sure balancebot's body angle is always at the reference balance position, which is $\alpha - \beta = 0$, so that balancebot will be stable whenever it is standing, moving, or turning. The input data is $\alpha - \beta$ and output is motor angular velocity (ω_{bl}, ω_{br}) to motor speed controllers.

3) Velocity controller: Outside the balance and heading controllers, the velocity controller (PID_v) controls balancebot's body velocity at the desired value. Instead of directly controlling the innermost motor speed, the velocity controller controls the balancebot's body velocity by adding a reference pitch angle (β) to the default balance position which is assumed to be the z-axis in Figure 1. To [illegible]e this given reference, the balancebot needs to accelerate or decelerate so that the desired speed can be reached. As the real velocity converges to the desired velocity, β will gradually reduce to zero and the balancebot can stay at a constant velocity.

4) Position controller: In the outermost layer is the position controller (PID_d). It makes sure the balancebot is always at the desired position. The input will be position information either given by the Optitrack in task 4 or the odometry in task 1 and 2. In the appendix table III, the parameters for standing at a fixed positon is given.

B. Odometry and Tracking

1) Parameters and equations of odometry model:

1) x_k denotes the position of balancebot with respect to the coordinate of x axis in kth calculation.
2) x_y denotes the position of balancebot with respect to the coordinate of y axis in kth calculation.
3) en_{avg} denotes the mean value of the data from two encoders after median filter in every calculation.
4) res denotes the number of slits per motor shaft revolution.
5) v denotes the velocity computed by linear acceleration from IMU.
6) d denotes the wheel diameter.
7) $ratio$ denotes the gear ratio of motor.
8) θ denotes the angle collected from IMU in Figure 1.
9) t_s denotes the sampling time step.

The equations to calculate odometry by encoder data only are:

$$\begin{aligned} \Delta x_k &= \frac{en_{avg}}{res \times ratio}\pi d cos(\theta) \\ \Delta y_k &= \frac{en_{avg}}{res \times ratio}\pi d sin(\theta) \\ x_{k+1} &= x_k + \Delta x_k \\ y_{k+1} &= y_k + \Delta y_k \end{aligned} \tag{1}$$

2) RTR planner: To make the balancebot follow the waypoints, an advanced version of "Rotate, Translate, Rotate" RTR planer is developed. Unlike common RTR planar, the balancebot will predict the direction before reaching the goal, so it will move smoothly instead of point-by-point. The detail will be discussed in this section.

a) Turning Algorithm: Since the tracking control needs to consider both the position and heading angles, decoupled position and heading angles controllers have dead zone, i.e. when the heading angle error is zero but the robot is still off the waypoint, the position control cannot change the heading angle. In this way, balancebot requires coupled position and heading angle controllers. To deal with this dead zone, we add a position error term to the heading angle control, so that the position control can also control the heading angle. For the detailed implementation, we create a prediction point which is $L = 0.8$ meters ahead the center of balancebot. In Figure 3, the bold black arrow is the path balancebot tracks. We use the distance from balancebot center to path and the distance from prediction point to path to calculate the ω_{hl} and ω_{hr} to motor speed controllers.

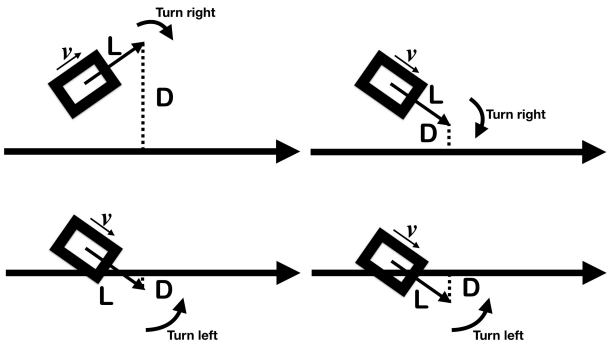

Fig. 3: Four cases of turning

In Figure 3, D denotes the distance from virtual head to path. If virtual head is at the left of path then D is positive, otherwise negative.

b) Tracking Algorithm: The path balancebot follows for navigation is composed of waypoints with step size less than 0.02 meters. To track a path

smoothly, balancebot needs to know which waypoint it has reached at every moment. To achieve this, we give balancebot a settled length (0.03m) shown as green line in Figure 4. **SP** and **GP** denote the start point and goal point of the line which balancebot is tracking at every moment respectively. Blue ball represents our balancebot. The line where green line's end lies on is the path that balancebot is tracking at every moment.

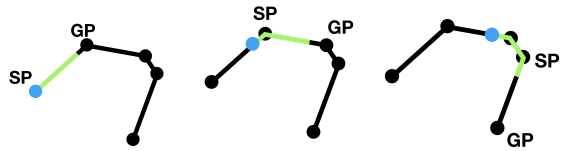

Fig. 4: tracking path

C. Path Planning

In this part, we develop a planner to make our robot move as desired. The planner is a waypoint generator based on navigation function algorithm. It decides what waypoints the robot should follow using the data from the optitrack.

1) Description and Rationale of Algorithm: To complete task 4, we use navigation function to find the path. Comparing to the common potential field method, the biggest advantage of navigation function is that it can avoid local minima.

Because our competition arena is an octagon, and we can consider it as a circle approximately, we use the planar sphere world model in our navigation function. All the obstacles in the world are defined as circles, and the robot is defined as a point in the world. Thus, to generate the configuration space of the robot, we add half of the maximum width of the robot to the radius of obstacles, and the radius values are shown in the following table.

TABLE I: Obstacle radius

Obstacle types	Radius (m)
Closed gate (whole gate)	0.475
Open gate (gate post)	0.170
Boundary of the arena	1.650

Since navigation function uses similar idea as potential field, repulsive and attractive forces are defined. There are two types of repulsive forces, the first one is from the boundary of the world[1], and the second one is from the obstacles in the world. They are defined using the following equations.

$$\begin{aligned} \beta_0(q) &= -d^2(q, q_0) + r_0^2 \\ \beta_i(q) &= d^2(q, q_i) - r_i^2 \end{aligned} \tag{2}$$

where q is current position, q_0 is the center of the sphere world, $d(\cdot, \cdot)$ is the Euclidean distance of two points, r_0 is the radius of the world, r_i is the radius of each obstacle. According to these two equations, β_0 is 0 at the boundary of the world, and always positive inside the world; β_i is 0 on the boundary of the obstacle, negative inside the obstacle, and positive in other places.

Thus, the total repulsive force is defined as the product of all repulsive forces.

$$\beta(q) = \prod \beta_i(q) \tag{3}$$

Then, the attractive force is defined as the following equation.

$$\gamma_K(q) = (d(q, q_{goal}))^{2K} \tag{4}$$

where q_{goal} is the desired position (the goal), K is an adjustable parameter in the navigation function, and it controls the significance of the attraction of the goal.

Finally, the potential of a point in the navigation function is defined as:

$$\phi(q) = \frac{d^2(q, q_{goal})}{(\gamma_K(q) + \beta(q))^{\frac{1}{K}}} \tag{5}$$

Using this equation, we can calculate the potential of every point in the world. Actually, we only need to calculate the potential of the current point q, the points quite close to current point in both x and y direction q_x and q_y, and use the partial derivative of potential on x and y to determine the next point.

$$\begin{aligned} \dot{\phi(q_x)} &= \frac{\phi(q) - \phi(q_x)}{\Delta x} \\ \dot{\phi(q_y)} &= \frac{\phi(q) - \phi(q_y)}{\Delta y} \\ v &= \frac{[\dot{\phi(q_x)}, \dot{\phi(q_y)}]}{|[\dot{\phi(q_x)}, \dot{\phi(q_y)}]|} \\ q_{next} &= q + \Delta d \cdot v^T \end{aligned} \tag{6}$$

where v is the normalized partial derivative vector, Δd is the step size.

With the previous equations in a loop, it can keep finding the next waypoint until it is less than $5mm$ away from the goal. The next thing to do is to design some goals for the navigation function using the gate positions.

Because we are given the position of left and right gates, and the passing direction is fixed (left gate is always on the left of the robot), we can design a 'in-point' where the robot stands before the gate, and a 'out-point' where the robot stands after passing the gate, and they are shown in the following Figure 5:

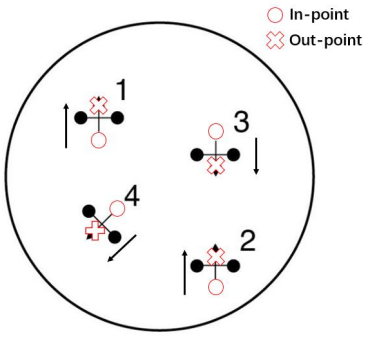

Fig. 5: Gate configuration example. Circle is 'in-point', cross is 'out-point'

The 'in-point' and 'out-point' are calculated as the following equations:

$$\begin{aligned} center.x &= \frac{gate_{left}.x + gate_{right}.x}{2} \\ center.y &= \frac{gate_{left}.y + gate_{right}.y}{2} \\ v_{dir}.x &= gate_{left}.y - gate_{right}.y \\ v_{dir}.y &= -(gate_{left}.x - gate_{right}.x) \\ in\_point.x &= center.x - (r_{gate} + 0.03)v_{dir}.x \\ in\_point.y &= center.y - (r_{gate} + 0.03)v_{dir}.y \\ out\_point.x &= center.x + (\frac{r_{gate}}{2} + 0.03)v_{dir}.x \\ out\_point.y &= center.y + (\frac{r_{gate}}{2} + 0.03)v_{dir}.y \end{aligned} \tag{7}$$

The robot moves from the 'in-point' to the 'out-point', and they move to the next 'in-point' until it has passed all the gates. Once the robot reaches the 'in-point', the corresponding gate will be open, so in the rest of the path-finding process, this gate is no longer considered, and only the gate posts are considered. In our competition, there are 4 gates in total, so there are 4 'in-points', 4 'out-points', and 1 final destination, 9 desired goals in total.

The algorithm for the whole path planner is the following:

Finally, we have a set of waypoints which the robot should follow to pass through the gates.

2) Implementation Considerations: When implementing this algorithm, there are some considerations when coding:

1) The term $\gamma_K(q) = (d(q, q_{goal}))^{2K}$ in equation 6 can be in extremely high order, the numeric value of the distance should be controlled in a reasonable range so that the result does not go beyond the 64-bit limit, and also computable for double precision number. Thus, meter is chosen to be the unit of distance.

Algorithm 1 Algorithm of path planner based on navigation function

```
calculate the 9 desired goals goals[9]
index = 0
for i = 1 to 9 do {find path to the i^th goal}
    q_goal = goals[i]
    for K = 4 to 25 do {try different K in navigation function}
        has_error = false
        count = 0
        while d(q, q_goal) > 0.005 do {q is not close enough to q_goal}
            calculate q_next {using equation (3-7)}
            waypoint[index + count] = q_next
            count++
            if (count > 1000) or (q_next is not a number) then {q stuck in a local minimal or unable to be solved}
                has_error = true
                break {try next K}
            end if
        end while
        if has_error = false then
            index+ = count
            break {if no error, the path is already found, move on to next goal}
        end if
    end for
end for
```

2) When calculate the next point q_{next}, the step size of 0.01m is chosen. It is accurate enough for this task, and under this step size, only about 200 waypoints are needed for each piece of path, which will not take much time for calculation.

3) Final Implementation: Then, to make the code run on our robot's operation system, we rewrite the MATLAB code into a C version. With the same gate position, the two program find the same path. The path generated by the C code is shown in the appendix. For fully automatic task 4, we only change the hard coded gate position into the value read from Optitrack, other code is exactly the same.

III. Results

A. Controllers

1) Left and right motor controller: In Figure 6, the Kp_m, Ki_m and Kd_m are tuned to show the underdamped and overdamped behavior of motor. Moreover, a 0.05 offset is added to the magnitude of dutycycle to cancel the stall torque created by friction. The stall torque is known as the dead zone of DC motor, which is shown in Figure 13 in appendix.

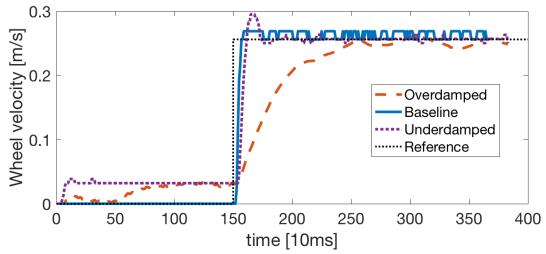

Fig. 6: DC motor's step response with baseline (Kp_m=1,Ki_m=7,Kd_m=0), underdamped (Kp_m=1,Ki_m=100,Kd_m=0) and overdamped (Kp_m=1,Ki_m=100,Kd_m=70) cases.

2) Heading and balance controller:

a) Heading controller: The heading controller itself is robust against Kp_h's variation because Kp_h only influences the time it takes to reach the desired angle as shown in Figure 7. However, when Kp_h is too large, high turning speed will cause the balance controller unstable as circled in Figure 7 when the Kp_h increases by 80%.

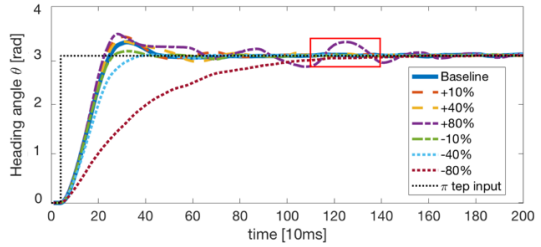

Fig. 7: Heading controller's performance for different Kp_i under π rad step input in time domain.

b) Balance controller: In Figure 8, the step response of balance controller for a 0.1 rad step input is given. The balancebot will fall at last for any body angle step input since the body speed reaches maximum and the body angle cannot be adjusted. The balance controller is robust against Kp_b variation as shown in the Figure 14 and 15 in appendix. The body angle for 0.1 rad step input is still under control when the Kp_b is increased by less than 100% or decreased by less than 50%.

3) Velocity controller: This behavior can be seen clearly in Figure 9 for a step response. In the Figure 16 and 17 in appendix, the step response of velocity controller against different Kp_v in time domain is shown. The velocity control is also robust against different Kp_v. Kp_v lower than baseline will not hurt balance control, so the balancebot can always stay balance while the velocity will converge to the reference velocity slower. The Kp_v

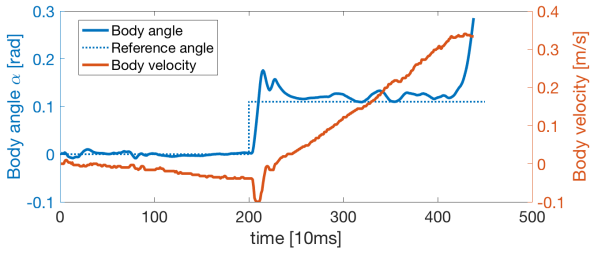

Fig. 8: Balance controller's response for 0.1 rad step input in time domain.

cannot be increased by more than 40%, or the abrupt velocity change will destroy the balance control.

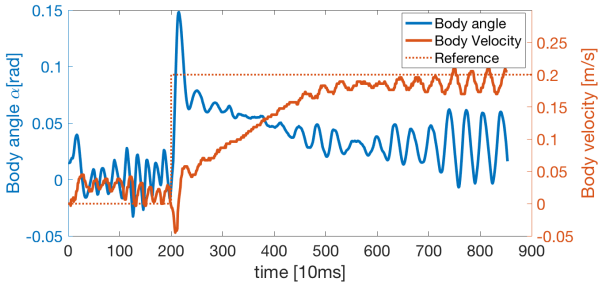

Fig. 9: Velocity controller's response for 0.2 m/s step input in time domain.

B. Odometry and tracking

1) RTR square-test result: By using our turn control algorithm, our balancebot can follow the 1m square. But with the statistic offset and the difference of motor in odometor, the error of square tracking is accumulating as time goes on.

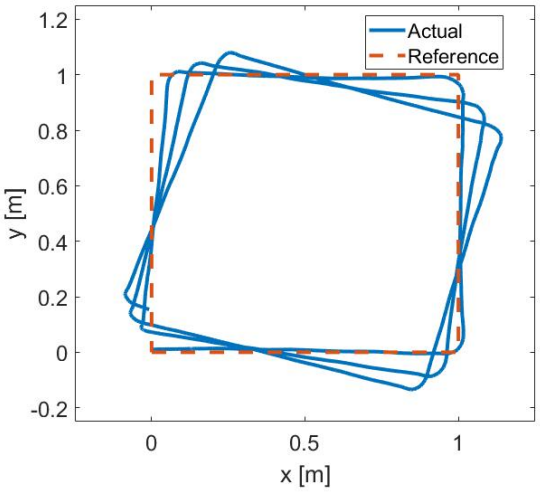

Fig. 10: Tracking performance with 1m width square

2) Validation of odometry model: To validate the odometry model, balancebot follows a sine-shape curve ($y = \frac{1}{3}sin(\frac{2\pi}{0.6}x)$ m) by the feedback from Optitrack and records the real-time position from Optitrack and odometry.

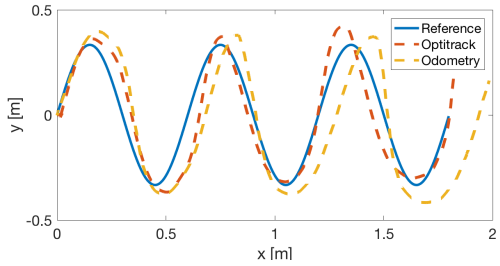

Fig. 11: Balancebot follows sine curve with optitrack feedback

The accuracy is shown by the curves above and the corresponding RMS error. The curves of Optitrack and reference show the tracking algorithm error. The RMS error between Optitrack and reference is 0.0481m. The curves of odometry and Optitrack show the error of odometry. RMS error between odometry and Optitrack is 0.1046m and the error will accumulate as the balancebot travels.

The noise from IMU accelerations is approximately Gaussian noise with $\sigma = 0.38m/s^2$. The histogram of noise data is shown in Figure 18 in appendix. The uncertainty from encoder is mainly the random peak but has been filtered out by median filter.

C. Path planning

To test the correctness of the algorithm, we first run a MATLAB simulation with the hard-coded gate position provided by instructor (in unit of meter):

$$
\begin{aligned}
Gate1 &: L : (0.305, 0.915) \quad && R : (-0.305, 0.915) \\
Gate2 &: L : (-1.220, 0.305) \quad && R : (-0.610, 0.305) \\
Gate3 &: L : (-0.610, -1.220) \quad && R : (-0.610, -0.610) \\
Gate4 &: L : (0.610, -0.305) \quad && R : (0.610, -0.915)
\end{aligned}
$$

The result is shown in the following Figure 12:

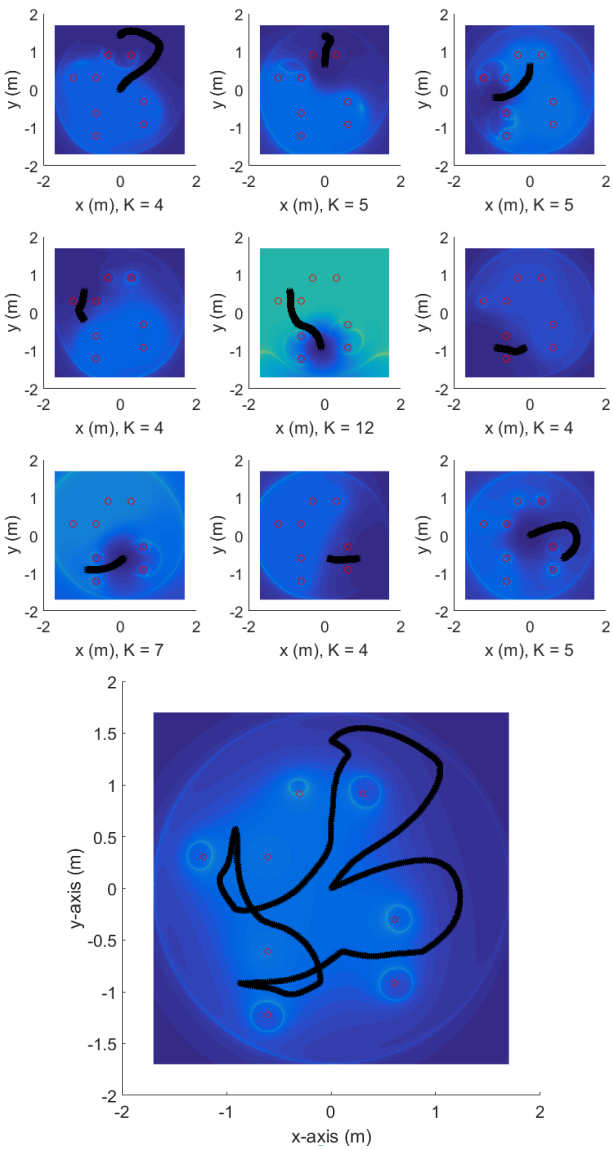

Fig. 12: 9 pieces of path and the whole path

In these two figures, the red circles are the gate posts and the black line is the path. In those 9 small plots, the blue-green background is the potential of every point on the map, calculated according to the goal and obstacles specifically for that step. Dark blue means the lowest potential, and green is the highest potential. So the goal always has the darkest blue, and the robot will move towards it. The K value in navigation function is labeled in each subplot.

APPENDIX

TABLE II: BOM

Part name	Number of parts
Beaglebone Green	1
Mobile Robotics Cape	1
Floureon LiPo battery 11.1V, 35C, 1500mAh	1
3s LiPo battery monitor	1
Polulu motor 25Dx50L, 20.4:1, 48 CPR encoder	2
Polulu motor driver MAX14870	2
REDCON satellite receiver 2.4GHz	1
MPU9250 IMU	1
EDIMAX EW-7811Un wifi adapter	1
Acrylic board 180x100 mm	2
90 mm metal pillar	2
120 mm metal pillar	4
150 mm metal pillar	2
80 mm diameter robot wheel	2
Triangular connector	4
Perpendicular connector	4
M3 screws	60

TABLE III: Parameter for controllers

Controller	Kp	Ki	Kd
Left motor	$1s/10^2$	$7s^2/10^4$	0
Right motor	$1s/10^2$	$7s^2/10^4$	0
Heading angle	1	$0s/10^2$	$0(10^2s^{-1})$
Balance angle	$2.5(10^2s^{-1})$	10	$6(10^4s^{-2})$
Velocity	$0.35s/10^2$	$0.05s^2/10^4$	0.2
Position	$3m^{-1}(10^2s^{-1})$	$1m^{-1}$	$10m^{-1}(10^2s^{-2}$

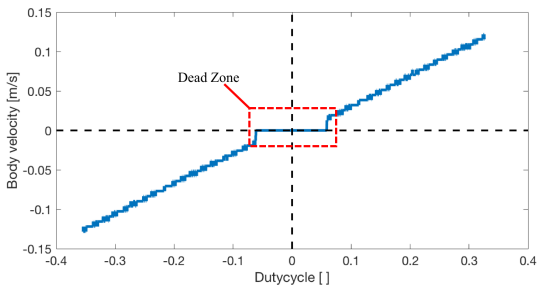

Fig. 13: Dead zone of DC motor

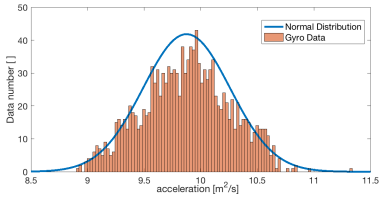

Fig. 18: Statistic noise distribution of IMU accelerations of forward direction of balancebot measured by making balancebot still

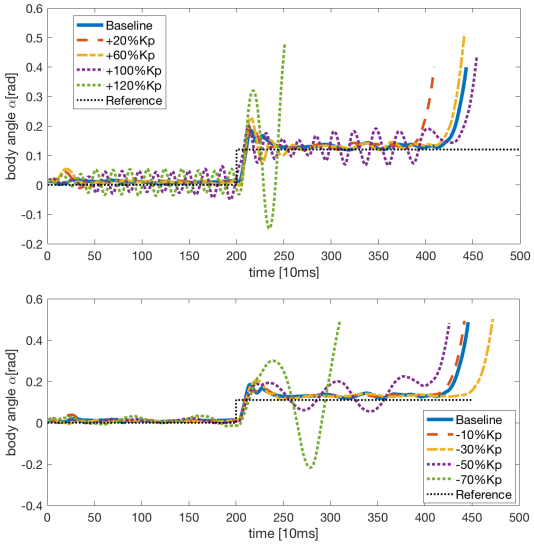

Fig. 14: (a)(b) Step response of body angle for balance control with 0.1 rad step input against different Kp_b

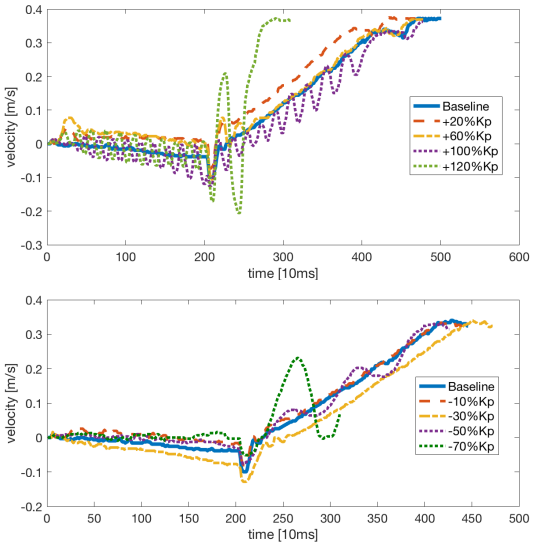

Fig. 15: (c)(d) Step response of body velocity for balance control with 0.1 rad step input against different Kp_b

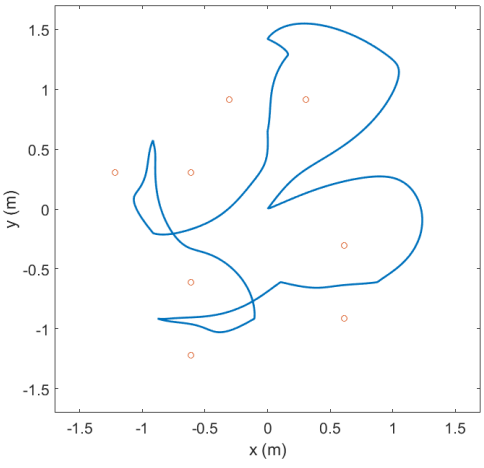

Fig. 19: Whole path found in C program.

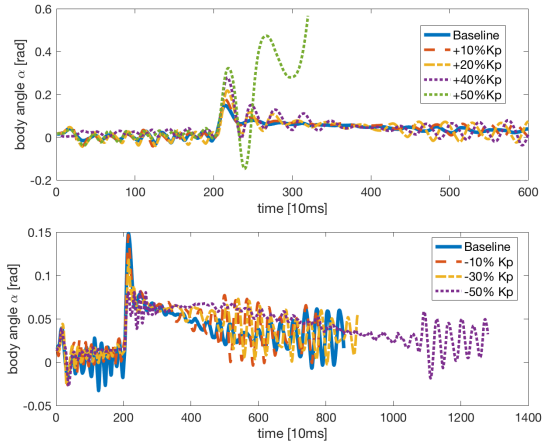

Fig. 16: (a)(b) Step response of body angle for velocity control with 0.1 rad step input against different Kp_b

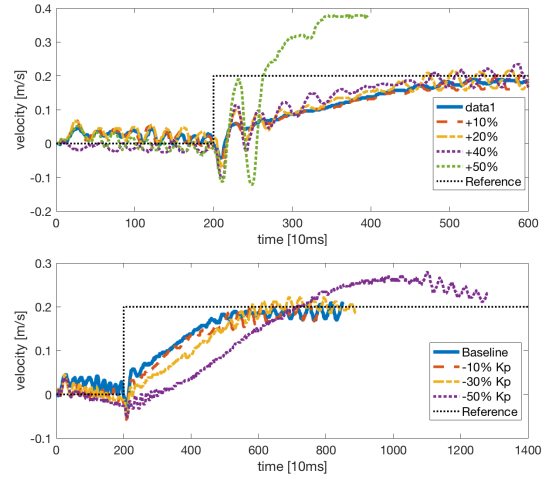

Fig. 17: (c)(d) Step response of body velocity for velocity control with 0.1 rad step input against different Kp_b

REFERENCES

[1] E. Rimon and D. E. Koditschek, "Exact robot navigation using artificial potential functions," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 5, pp. 501–518, Oct 1992.