Team 4 ROB 550 Balancebot Report

Bo Xue, Hanyang Li, Zhongdong Liu

***Abstract*—In this Balancebot Lab, we successfully implemented the control over a two-wheeled inverted pendulum based on Beaglebone embedded system with Robotic Control Library in C. Our balancebot was able to integrate data from sensors like IMU and wheel encoders to perform PID control, full state feedback control, RC manual control, odometry & motion control, and trajectory following.**

I. Introduction

BALANCEBOT is a typical self-balancing dynamic system. How to create appropriate controllers to keep it balance is one of the most important goals. In this project, controllers are implemented to keep the balancebot balance. In order to achieve it, creating and tuning PID and LQR controllers are all necessary. By implementing effective controllers and using odometry, the balancebot can also move to positions as we want. By using optitrack we can get the position and orientation of the balancebot in the workspace. The lab results indicate that our balancebot can successfully keep balance and achieve specific position and orientation automatically as well as under manual control.

II. Methodology

A. Motor Measurement

We used following equations to calculate motor parameters,

$$V = Ri + L\frac{di}{dt} + K\omega_{shaft} \tag{1}$$

$$\tau = J\frac{\omega_{wheel}}{dt} + b\omega_{wheel} \tag{2}$$

where V is the applied voltage, which can be read on battery monitor. R is the coil resistance, which is measured using the multimeter. i is the current, which can be read using function `rc_motor_read_current()`. L the coil inductance, K is motor constant. ω_{shaft} and ω_{wheel} is shaft angular velocity, τ is the motor torque, J is the moment of inertia, and b is the friction coefficient. Since we set the duty as 1, the motor rotate at a static speed. So the current and the angular velocity does not change with time. Equations can be simplified as follows.

$$V = Ri + K\omega_{shaft} \tag{3}$$

$$\tau = b\omega_{wheel} \tag{4}$$

The motor torque, motor constant, and the current have relationship as follows.

$$\tau = Ki \tag{5}$$

Also, the angular velocity of motor and shaft have the relationship as

$$\omega_{wheel} = \omega_{shaft} \cdot gearbox \tag{6}$$

where $gearbox$ is 20.4 according to manufacturer. Then the equations can be modified as

$$V = Ri + K\omega_{wheel}/gearbox \tag{7}$$

$$Ki = b\omega_{wheel} \tag{8}$$

The motor's angular velocity is calculated by following equation,

$$\omega_{wheel} = \frac{2\pi(E - E_{prev})}{gearbox \cdot resolution \cdot DT} \tag{9}$$

where DT is constant time interval, which is 0.01s for this lab. E is the current average encoder reading of two wheels, and E_{prev} is the average reading DT ago. The product of gearbox and resolution is 979.62 according to manufacturer, which is encoder counts per motor shaft revolution. Encoder reading is obtained using `rc_encoder_eqep_read()`. Then K is computed by

$$K = \frac{(V - Ri)gearbox}{\omega_{wheel}} \tag{10}$$

Knowing K, b is calculated by

$$b = \frac{Ki}{\omega_{wheel}} \tag{11}$$

Also, the stall torque, denoted by τ_s can be achieved by

$$\tau_s = \frac{KV}{R} \tag{12}$$

The no-load speed:

$$\omega_{NL} = \frac{KV}{K^2 + Rb} \tag{13}$$

To determine the inertia of the motor armature, shaft and gearbox, denoted by I_M, we need to have the time constant of step response.

$$\tau = \frac{I_M}{b} \tag{14}$$

where τ is the time constant (different from the torque), which is 66.7% of the rise time of the step response. With b and τ, we can achieve the inertia.

B. Balance Control

We used two different feedback controllers for balancing our balancebot, which were PID controller and LQR controller. Also, we implemented another PID controller to control the heading of the balancebot.

1) PID controller: The balancebot system model has one input, which is the motor command denoted by τ, and two outputs, which are body angle and wheel position, denoted by θ and ϕ.

We built two control loops to implement the body angle and wheel position control. The inner loop takes the motor command as the input, and the output was the body angle. The transfer function in Laplace domain for the inner loop:

$$G_1(s) = \frac{m_b R_\omega L + I_\omega + (m_\omega + m_b)R_\omega^2}{(-(I_\omega + (m_\omega + m_b)R_\omega^2)(I_b + m_b L^2)s^2 + m_b g L(I_\omega + (m_\omega + m_b)R_\omega^2)} \tag{15}$$

where m_b and m_ω are mass of the body and the wheel, R_ω is the radius of the wheel, L is the length from end to center of mass of the body, I_b and I_ω are inertia of the body and the wheel, g is the gravity acceleration. By applying parameters we measured, the inner loop transfer is expressed as follows.

$$G_1(s) = \frac{-140s}{s^3 + 13.21s^2 - 91.25s - 474.8} \tag{16}$$

We used a PID controller to implement the body angle control. It took the output of the inner loop, which was the body angle, as the feedback, and gave the motor torque as the output. The transfer function of the inner loop PID controller is as follows.

$$D_1(s) = K_P + \frac{K_I}{s} + K_D s = \frac{K_D s^2 + K_P s + K_I}{s} \tag{17}$$

TABLE I: Parameters of PID Controllers

State	K_P	K_I	K_D
θ	3.0	14.0	0.05
ϕ	0.005	0.0003	0.003
γ	0.01	0.0003	0.003

By applying parameters of the controller in the Table IV, the transfer function $D_1(s)$ of inner loop controller can be expressed as

$$D_1(s) = \frac{0.05s^2 + 3s + 14}{s} \tag{18}$$

Then we can build outer loop to control the position of our balancebot. The outer loop used the body angle, which was the output of the inner loop as the input, and gave the wheel position as the output. The transfer function in Laplace Domain of outer loop is expressed as follows.

$$G_2(s) = \frac{(-I_b + m_b R_\omega L)s^2 + m_b g L}{(I_\omega + (m_\omega + m_b)R^2 + m_b R_\omega L)s^2} \tag{19}$$

By applying parameters we measured, the transfer function can be expressed as follows.

$$G_2(s) = \frac{-3.03s^2 + 144.8}{s^2} \tag{20}$$

We used another PID controller to control the wheel position. It took the difference of reference wheel position and the outer loop output phi as the input. The output of the controller was the reference body angle, which was used to compute the body angle error. Parameters of outer controller are also shown in Table IV. Applying parameters, the transfer function of outer loop, $D_2(s)$, can be expressed as follows.

$$D_2(s) = \frac{0.003s^2 + 0.005s + 0.0003}{s} \tag{21}$$

The full control system of PID control is shown in Figure 1,

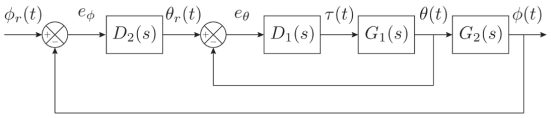


Fig. 1: Full Control System of PID Control

where $\phi_r(t)$ was the reference wheel position. e_ϕ and e_θ were the error of wheel position and body angle, which were inputs of two PID controllers respectively. $\theta_r(t)$ and $\tau(t)$ were reference body angle and motor command, which were outputs of two controllers respectively. $\theta(t)$ and $\phi(t)$ were body angle and wheel position of the balancebot, which were outputs of inner and outer loop. They were the states of the system, and could be achieved by reading the Euler angle along x axis from the IMU data, `mpu_data.dmp_TaitBryan[TB_PITCH_X]`, and the encoder, which is

$$\phi = \frac{2\pi(E - E_{prev})}{gearbox \cdot resolution} \tag{22}$$

Then we can control these two states by implementing the PID controller.

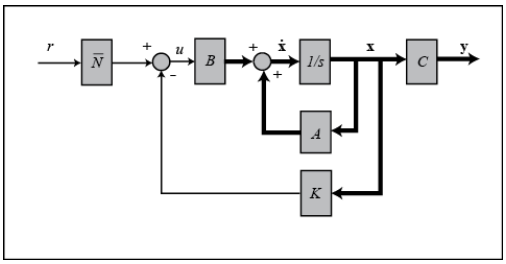


Fig. 2: The control system of the full state feedback controller (LQR)

2) LQR controller: A state space model of a linear time invariant system has the form:

$$\dot{x} = Ax + Bu \tag{23a}$$

$$y = Cx + Du \tag{23b}$$

For our balancebot system, we care about 4 states, which are the body angle θ, body angle velocity $\dot{\theta}$, wheel angle ϕ, and wheel angle velocity $\dot{\phi}$ respectively. Therefore, we take the state vector to be

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \\ \phi \\ \dot{\phi} \end{bmatrix} \tag{24}$$

θ is the values given by the DMP sensor, can be read using `mpu_data.dmp_TaitBryan[TB_PITCH_X]`, $\dot{\theta}$ is given by the Gyro, can be read using `mpu_data.gyro[0]`, ϕ can be calculated from the values given by the wheel encoder. And $\dot{\phi}$ can be calculated by the difference of the current ϕ and the previous ϕ in each 0.01 second time interval.

$$\phi = \frac{2\pi(E - E_{prev})}{gearbox \cdot resolution} + \theta \tag{25}$$

The θ is added to ϕ because the wheel encoders measure the relative wheel angle to the body, and the ϕ measure the actual angle that wheels rotated to the world frame, thus it need to be added θ as an offset. On the other hand, the gain matrix K is calculated from the MATLAB simulation of the state space model using MATLAB function `dlqr(A, B, Q, R)`, where Q is the positive matrix that represent state cost, in this problem, is in the form of

$$Q = \begin{bmatrix} q_\theta & 0 & 0 & 0 \\ 0 & q_{\dot{\theta}} & 0 & 0 \\ 0 & 0 & q_\phi & 0 \\ 0 & 0 & 0 & q_{\dot{\phi}} \end{bmatrix} \tag{26}$$

and R is a positive matrix that represent control effort cost, and in this problem is a positive scalar. They are used to minimize the cost function

$$J = \int_0^\infty (x^T Q x + u^T R u) dx \tag{27}$$

In our problem, we take Q matrix as

$$Q = \begin{bmatrix} 50 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix} \tag{28}$$

and set $R = 500$, and obtained K matrix such that

$$K = \begin{bmatrix} K_1 \\ K_2 \\ K_3 \\ K_4 \end{bmatrix} = \begin{bmatrix} -5.5840 \\ -0.7489 \\ -0.3705 \\ -0.1755 \end{bmatrix} \tag{29}$$

For the zero reference problem, the duty cycle u is calculated as

$$u = K_1\theta_e + K_2\dot{\theta}_e + K_3\phi_e + K_4\dot{\phi}_e \tag{30}$$

where θ_e, $\dot{\theta}_e$, ϕ_e, $\dot{\phi}_e$ are the state errors deviate from the references.

When the reference states are not 0, we need to implement the precompensator $\overline{N}$ as shown in Figure 2. This is because the full-state feedback system does not compare the output to the reference; instead, it compares all states multiplied by the control matrix ($K\mathbf{x}$) to the reference. Thus, the output is not to be equal to the commanded reference. To obtain the desired output, we will need to scale the reference input so that the output does equal the reference in steady state. This can be done by introducing a precompensator scaling factor called $\overline{N}$. In MATLAB, the system was modeled as `ss(A_dc,B_dc*Nbar_d,C_dc,D_dc,DT)`.

For the case of only having wheel reference position, since we don't care about other states, we can take the precompensation $\overline{N} = K_3 = -0.3705$.

3) Heading controller: The heading information of the balancebot, denoted by γ could be achieved by reading the Euler angle along the z axis from the IMU data. To control the heading of our balancebot we implement a P controller using difference between reference heading and γ as the input. The controller parameters were shown in Table IV, so the transfer function could be described as follows.

$$D_3(s) = \frac{0.003s^2 + 0.01s + 0.0003}{s} \tag{31}$$

Then the motor command of full PID controller can be expressed as follows,

$$\tau_L = u_1 - u_3 \tag{32a}$$

$$\tau_R = u_1 + u_3 \tag{32b}$$

where u_1 is the out loop output and u_3 is the heading control output. τ_L and τ_R are motor commands of left and right wheels.

In the coding, I give the error of heading to the PID controller, named `gamma_error`. And `gamma_error`

is equal to `mb_state.gamma`, the heading angle measured by the gyro, plus the impulse step command, which steps from zero to one and does not change. The PID output is just the u_3 above.

The parameters we used are the PID gains, step impulse value, and the heading value measured by the gyro. The PID gains are shown before in Table IV.

C. Odometry

Our team used three different models to update position estimate.

1) Motion Model: By building the motion model, the motion of the robot can be specified as a rotation around a center point. By assuming the centers of two wheels and the rotation center are always in a straight line, which means that the angle swept by each wheel must be the same, and the distance travelled along path, δd, is equal to the arc length for small changes, the motion of robot can be described using the arc length and the angle of the rotation. First, the angle of the rotation, which is also the change of heading $\Delta\gamma$, is determined by the travelled distance of two wheels,

$$\Delta\psi = \frac{\Delta s_L - \Delta s_R}{b} \tag{33}$$

where b is the wheel base. Also, the distance travelled along the path can be expressed by

$$\Delta s = \frac{\Delta s_L + \Delta s_R}{b} \tag{34a}$$

$$\Delta d \approx \Delta s \tag{34b}$$

as a result, the new position of robot can be expressed as follows.

$$\Delta x = \Delta d cos(\psi + \Delta\psi/2) \tag{35a}$$

$$\Delta y = \Delta d sin(\psi + \Delta\psi/2) \tag{35b}$$

Then the new position and orientation, denoted by p' can be expressed as follows.

$$p' = \begin{bmatrix} x' \\ y' \\ \psi' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\psi \end{bmatrix} \tag{36}$$

To implement the motion model, all we need to know are Δs_L and Δs_R. By reading the encoder readings of two wheels, the travelled distances of two wheels can be expressed as,

$$\Delta s_L = \frac{2\pi(E_L - E_{Lprev})}{gearbox \cdot resolution} \tag{37a}$$

$$\Delta s_R = \frac{2\pi(E_R - E_{Rprev})}{gearbox \cdot resolution} \tag{37b}$$

where E_L and E_R are current encoder readings of two wheels, E_{Lprev} and E_{Rprev} are encoder readings 0.01s ago. The resolution is 979.62, which is encoder counts per motor shaft revolution. Then the motion model can be achieved by only using the encoder reading.

2) Velocity Model: Same as motion model, the model of robot is regarded as rotation around the center point, where the radius of rotation is defined by R and the rotation center is defined by C_R. Define the angle velocity of rotation as ω, the linear velocity of each wheel can be expressed as follows,

$$v_R = \omega(R + b/2) \tag{38a}$$

$$v_L = \omega(R - b/2) \tag{38b}$$

where v_R and v_L are linear velocity of left and right wheels. Then we can compute R, ω, and C_R,

$$R = \frac{b(v_R + v_L)}{2(v_R - v_L)} \tag{39}$$

$$\omega = \frac{v_R - v_L}{b} \tag{40}$$

$$C_R = [C_{Rx}\ C_{Ry}] = [x - Rsin\theta\ \ y + Rcos\theta] \tag{41}$$

Then the new position can be expressed as follows.

$$p' = \begin{bmatrix} cos(\omega\Delta t) & -sin(\omega\Delta t) & 0 \\ sin(\omega\Delta t) & cos(\omega\Delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - C_{Rx} \\ y - C_{Ry} \\ \psi \end{bmatrix} + \begin{bmatrix} C_{Rx} \\ C_{Ry} \\ \omega\Delta t \end{bmatrix} \tag{42}$$

To implement this model, we need to compute the linear velocity of both two wheels, v_L and v_R, then we can compute R and ω. Since the encoder readings were updated every DT, then the velocity can be expressed as follows.

$$v_L = \Delta s_L / DT \tag{43a}$$

$$v_R = \Delta s_R / DT \tag{43b}$$

As a result, we can implement the velocity model by only using the encoder reading. However, when calculating R, if two wheels has the same velocity, R is infinity so we can't use this model in this situation. There are two solutions. We can set a very large R when two wheel has the same velocity, or we can just use the motion model for this specific situation. We used the second method to implement the velocity model, which were actually the mix of two models.

3) Gyro-fused Model: Since we can read the data from gyro to achieve the angular velocity of the rotation. We can use directly use it instead of computing it from linear velocity, so the gyro-fused model we used was the same as velocity model except the method we used to get ω. In this model we still need encoder reading to calculate the rotation radius R.

4) High level controller based on odometry: Using motion model of odometry, we can achieve the distance travelled along the path Δd and the heading change $\Delta\psi$ in every time interval of 0.01s. Given initial values, we can know states of travelled distance d and heading psi,

$$d' = d + \Delta d \tag{44}$$

$$\psi' = \psi + \Delta\psi \tag{45}$$

where d' and psi' are travelled distance and heading angle of new state. By replacing the states of PID controller, which are ϕ and γ, with d and psi, the performance of the control was improved. When implementing the high level controller, we did not modify the parameters of PID controller and they are shown in Table IV.

5) Parameters: The parameters we used in this part are the baseline, wheel diameter, encoder resolution, interval time to calculate the speed. For this part, we got the nominal values of the baseline and wheel diameter by simply measure them using vernier caliper. The values we get are: $baseline = 0.206m$, $wheel\_diameter = 0.0805m$. The encoder resolution is gotten by looking up the user manual and is 979.62. Since we call the odometry function every time there is a new IMU data and the IMU data is selected by interrupt at the frequency $100HZ$, the time interval should be 0.01 seconds.

III. Results

A. Motor Control

The mass and moments of inertia around x, y, z axis are as follows

TABLE II: Physical parameters of Balancebot

Parameter	Description	Value
m	body mass of the balancebot	1.064 kg
J_{xx}	moment of inertia around x	$3.755\times10^{-4}Kgm^2$
J_{yy}	moment of inertia around y	$6.652\times10^{-4}Kgm^2$
J_{yy}	moment of inertia around z	$5.234\times10^{-4}Kgm^2$

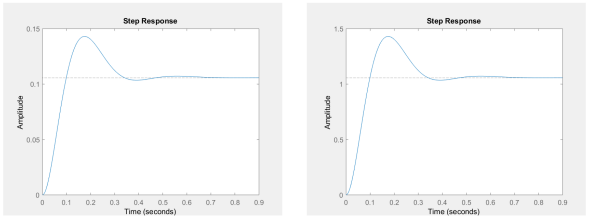
The motor parameters are

TABLE III: Parameters of Motor Measurement

Parameter	Left	Right
R	6.1Ω	5.9Ω
K	0.278	0.281
τ_s	0.56N·m	0.54N·m
b	2.82×10^{-5}	2.79×10^{-5}
ω_{NL}	42.76rad/s	42.56rad/s
I_M	$3.36\cdot10^{-6}kg\cdot m^2$	$3.35\cdot10^{-6}kg\cdot m^2$

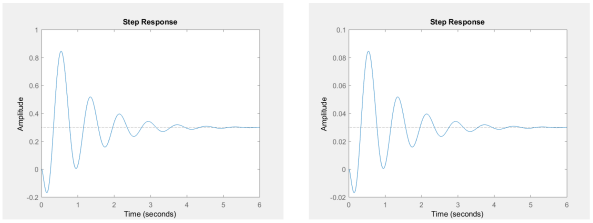
B. Balance Control

1) PID controller: By implementing the PID controller, our balancebot could remain balance around the initial position. However, we found that the balancebot could not remain balance well when the reference body angle was 0. By setting a body angle offset of 0.03, the performance was better. When tuning the PID controller, we found that the settling time was too long. To decreasing the settling time, we used 50 times of the wheel position error as the input of the outer loop, and 10 times of the body angle error as the input of outer loop. By adding the input, the control would be more aggressive and states could reach the reference value faster. First, we applied parameters on the Matlab file and got the simulated step response for our model, which is shown in Figure 3 and Figure 4. Simulated response shows that there was a small steady state error of our model.



(a) (b)

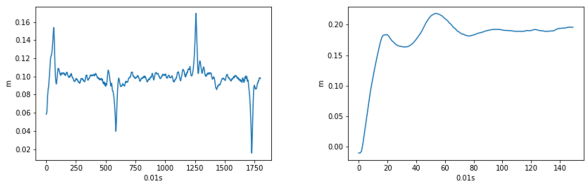
Fig. 3: (a) Simulated step response to a change of 0.1 rad in reference body angle; (b) Simulated step response to a change of 1 rad in reference body angle



(a) (b)

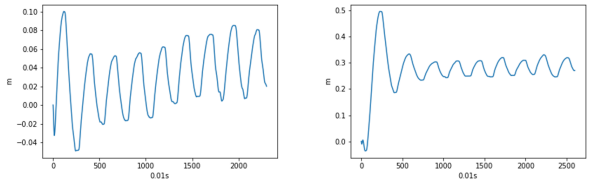
Fig. 4: (a) Simulated step response to a change of 0.03m in reference wheel position; (b) Simulated step response to a change of 0.3m in reference wheel position

Then we got the actual step response for the balancebot. We chose reference body angle as 0.1 rad and 0.18 rad because if the reference body angle was larger than 0.18, the balancebot would lose balance immediately so we could not get enough data. When the reference angle was set as 0.15 rad, the balancebot could move forward for about 2 meters before it lost balance. The step response of actual balancebot are shown in Figure 5 and 6.



(a) (b)

Fig. 5: Actual balancebot step reponse of PID controllers: (a) actual step response to a change of 0.1 rad in reference body angle; (b) actual step response to a change of 0.18 rad in reference body angle

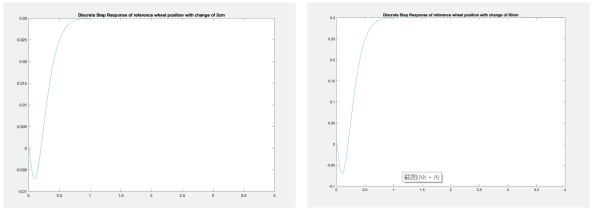


(a) (b)

Fig. 6: Actual balancebot step reponse of PID controllers: (a) actual step response to a change of 0.03m in reference wheel position; (b) actual step response to a change of 0.3m in reference wheel position

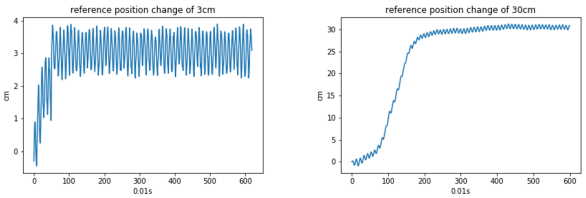
2) LQR controller: We measured all the physical parameters of balancebot and motors and used them in MATLAB to build a state space model, and then obtained the gain matrix K, which is used to implement the LQR controller in `balancebot.c` code. Before implement on the actual robot, we also ran simulations in MATLAB to see the step response of the reference changes in body velocity and wheel position.

The step response of the simulated Balancebot to a change of position in reference wheel position is shown as Figure 7, while the actual response is shown as Figure 8.



(a) (b)

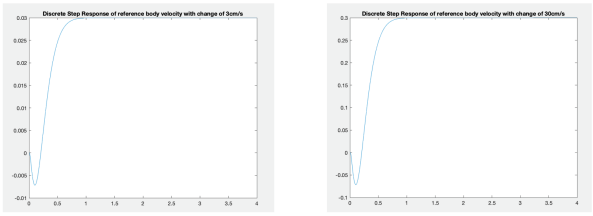
Fig. 7: (a) the step response of the simulated Balancebot to a change of 3cm in reference wheel position; (b) the step response of the simulated Balancebot to a change of 30cm in reference wheel position



(a) (b)

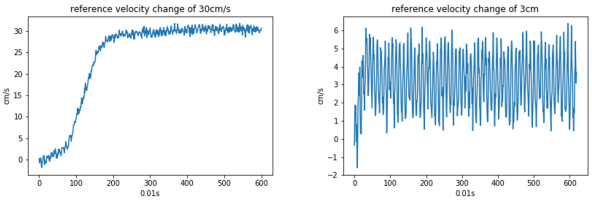
Fig. 8: (a) the step response of the actual Balancebot to a change of 3cm in reference wheel position; (b) the step response of the actual Balancebot to a change of 30cm in reference wheel position

The step response of the simulated Balancebot to a change of velocity in reference body velocity is shown as Figure 9, while the actual response is shown as Figure 10.



(a) (b)

Fig. 9: (a) the step response of the simulated Balancebot to a change of 3cm/s in reference body velocity; (b) the step response of the actual Balancebot to a change of 30cm/s in reference body velocity



(a) (b)

Fig. 10: (a) the step response of the actual Balancebot to a change of 30cm/s in reference body velocity; (b) the step response of the actual Balancebot to a change of 3cm/s in reference body velocity

3) Heading controller: After giving the balancebot a impulse heading command, we get a step response with regard to the input heading command. The goal heading is to turn 1 rad. Step response of heading is shown below in Figure 11:

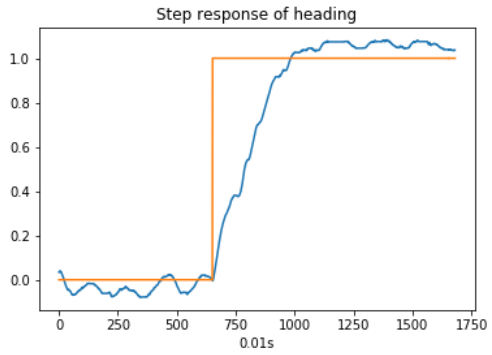


Fig. 11: Step response of heading. Yellow curve is the step impulse command. Blue curve shows the step response of heading. The horizontal axis is time(s), the vertical axis is the heading value

From the curve we can see that the balancebot has an immediate response to the step impulse. The rise time is about 3.5 seconds. The overshoot is about 0.08 rad. Settling time is about 9.5 seconds. The steady state value is about 0.03 rad.

C. Odometry

1) Performance of odometry and gyro fused odometry: There are two ways to implement the odometry: encoder-only odometry and gyro fused odometry. As illustrated before, the former takes only the encoder data to control the turning, while the latter takes the gyro data and encoder data to do the turning. In our experiment, we got results for both the methods and find that the gyro fused method is much less accurate than the encoder-only method. Results shown in Figure 12 and Figure 13:

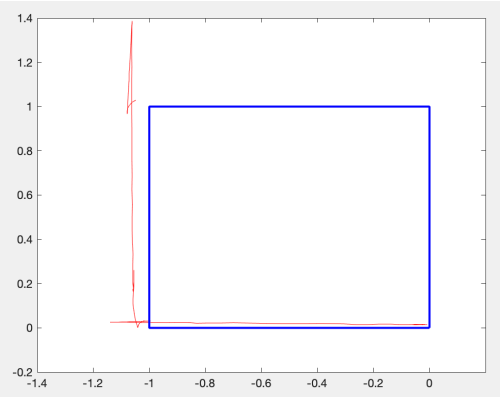


Fig. 12: Performance of the odometry with encoder only. The blue square is the reference where the balancebot thinks it is, while the red curve shows the ground of the movement

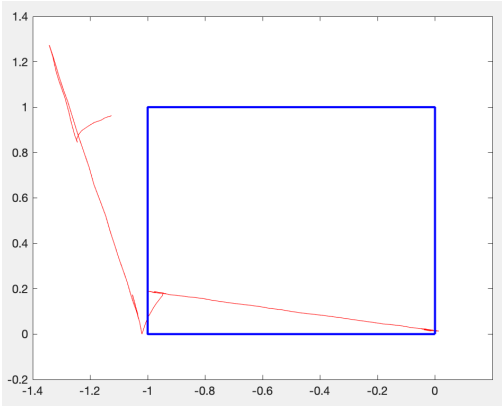


Fig. 13: Performance of the odometry fused gyro. The blue square is the reference where the balancebot thinks it is, while the red curve shows the ground of the movement

From the figures above, we can know that the gyro fused odometry has very bad performance comparing to the encoder-only method. Thus we choose to use the encoder-only odometry for all the control tasks.

2) Validating the odometry model: In order to find the actual value of the odemetry model parameters, we should update the parameters based on the initial values that we measured. However, in the experiment, we find that our initial values are accurate enough to get the good performance. In our experiment of letting the balanceot turning a 1 by 1 square map for one time, we get an relatively accurate performance(shown below in Figure 14). Thus if we implement the updating process of these parameters, the result would be similar to the one we have already gotten. So we just use the directly measured parameters instead of updating them.

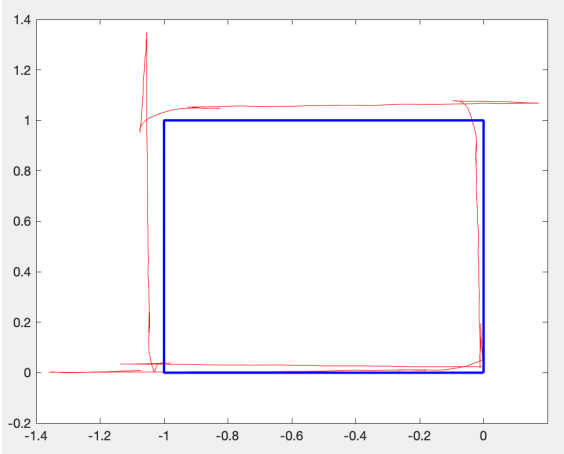


Fig. 14: Performance of the one-time square using the parameters that are directly measured. The blue square is the reference where the balancebot thinks it is, while the red curve shows the ground of the movement

D. Path Planning

After implementing the RTR method on the square routing task, we get the result shown in Figure 15:

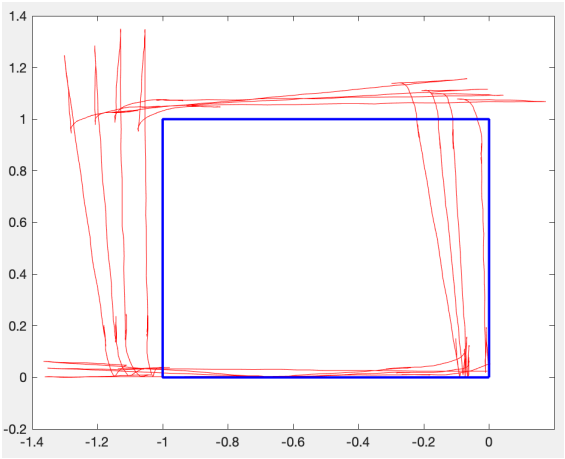


Fig. 15: Performance of square routing task. The blue square is the reference where the balancebot thinks it is, while the red curve shows the ground of the movement

The error of our implement is calculate using the equation below:

$$error_{process} = \sqrt{\Delta x^2 + \Delta y^2} \tag{46}$$

while Δx and Δy satisfies:

$$\begin{aligned} \Delta x &= \frac{1}{N}\sum_{i=1}^{N} x_i - X_i \\ \Delta y &= \frac{1}{N}\sum_{i=1}^{N} y_i - Y_i \end{aligned} \tag{47}$$

x_i and y_i are the ground truth point for every selected points, X_i and Y_i are the closest point on the goal square to the present ground truth point. After calculation, we get the error value of $0.8224m$. For the error between the final point and the goal point, we calculate using the following:

$$error_{end} = P_{start} - P_{stop} \tag{48}$$

while P_{start} is the start point position, P_{stop} is the stop point position. Thus we got the $error_{end}$ at x axis of $-0.097m$, the $error_{end}$ at y axis of $0.127m$, the distance of the $error_{end}$ of $0.1598m$ the heading error of 0.0678 rad.

IV. Discussion

A. Balance Control

1) PID controller: Our PID control worked well according to the step response plots and performance in Event 1. However, there were some problems. First, since the Euler angle along the x axis may not be zero at the initial position, the balancebot would turn a small angle after we ran the program. We fixed it by using ψ from the odometry as the input of the heading controller. Second, the settling time of our controller was still larger than expected even though we multiplied the error used for the PID controller. It indicates that our PID parameters, especially K_P and K_I are not appropriate.

2) LQR controller: Our LQR controller was successfully implemented to balance the robot. However, comparing to the PID controller, the LQR is obviously not as good as it. The problems are as follows. Firstly, unlike PID controller smoothly control the balancebot, it's oscillating frequently while balancing. In theory, this is due to the derivative terms like $\dot{\theta}$ and $\dot{\phi}$ contain high frequency components. In order to solve this oscillation issue, we will need to pass $\dot{\theta}$ and $\dot{\phi}$ into a low pass filter. Secondly, when the balancebot was tripped over, the motor duty cycle will be 1 or -1 forever, and can hardly balance again. Therefore, the LQR supports relatively small range of body angle balancing.

3) Odometry: The odometry controller was not accurate enough so we did not get a good performance on the square routing task. This may because the gains of the PID are not accurate enough and there are some inaccurate of the odometry parameters.

References

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

[2] J. Borenstein and L. Feng, "Gyrodometry: A new method for combining data from gyros and odometry in mobile robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1. IEEE, 1996, pp. 423–428.

[3] ——, "Measurement and correction of systematic odometry errors in mobile robots," *IEEE Transactions on robotics and automation*, vol. 12, no. 6, pp. 869–880, 1996.

[4] K. J. Aström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

Appendix

TABLE IV: MATLAB simulation modeling parameters

Parameter	Description	Value
DT	time interval	0.01s
m_b	body mass	1.062kg
m_ω	wheel mass	0.022kg
R_ω	wheel radius	0.04m
L	center to end length	0.14m
I_b	body inertia	0.00376 kg·m^2
I_ω	wheel inertia	0.0024 kg·m^2
g	gravity acceleration	9.81kg
R	coil resistance	6.0Ω
K	motor constant	0.28
V	applied voltage	12.0V
τ_s	stall torque	0.56N·m
b	friction coefficient	2.8*10^{-5}
ω_{NL}	no-load speed	42.76rad/s
I_M	inertia of armature, shaft, and gearbox	3.36·$10^{-6}kg \cdot m^2$