ROB 550 Balancebot Report

Manas Jyoti Buragohain, Xi Lin, Ziyou Wu
{manasjb, bexilin, wuziyou}@umich.edu

***Abstract*—We document the process of controlling a two-wheeled self-balancing robot by remoter controller, and do specific tasks: drag race and drive square autonomously. We first design PID controllers which allow the robot to maintain and regain balance when subjected to perturbations. Then we implement odometry and trajectory planning algorithm to track a given reference trajectory.**

Keywords: Self Balancing Robot, PID Control, Odometry

I. Introduction

In this lab we designed and implemented a self-balancing robot. In essence, the robot is an inverted pendulum on two wheels which is a naturally unstable system. A schematic drawing of balancebot is shown in figure 1. We implement cascaded PID filters which actively work towards maintaining equilibrium while providing it with navigational capabilities. We use the feedback from the 48 CPR encoder on the DC motor and linear and angular velocities from the 9DOF MPU9250 IMU to drive the PID loops.

Section II describes the methodology behind the robot, including motor model, controller implementation and trajectory follower. Section III shows and discusses the performance of the controller and trajectory follower. We plot step responses of each controller when subjected to a external offset and validate odometry model while testing its accuracy.

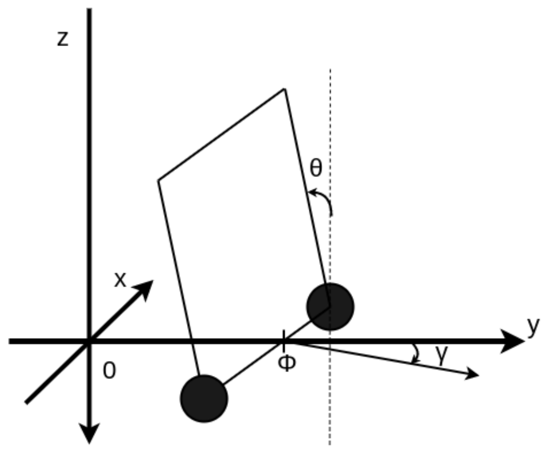

Fig. 1. Balancebot schematic model. The state of the balancebot can be described by body angle θ, position ϕ and orientation γ.

II. Methodology

A. Motor Model

We follow the step provided in [1] to measure the motor parameters. We first measure the motor resistance r for each motor using the multimeter. We then use the provided binary $measure\_motors$ to evaluate the No Load Speed ω, No Load Current A, Motor Constant K, Stall Torque τ, Shaft Friction f_s and Shaft Inertia I_s. We record our observation in Table V.

B. Controller

The model of the balancebot as presented in Fig 1 is an unstable system. While the model has a equilibrium state in the upright position, any disturbance to the bot would drive it away from the equilibrium state rendering it unable to recover its balance. Hence we design a set of cascaded PID controllers to control the position and orientation of the balancebot, while keep its balance.

We use three PID controllers. More specifically, we use two parallel control loops, one for position and another for orientation. The position controller is a successive loop controller with position controller in the outer loop and body angle controller in the inner loop. The orientation controller is a simple PID controller running parallel to position controller. The final motor command is given based on the output of both position and orientation controllers. The detailed flow chart for the whole system is in Fig.2. We will describe position, body angle and heading controllers in detail in the following sections. A summary of PID gains is in table I.

TABLE I
PID GAINS SUMMARY

controller	K_p	K_i	K_d
Body angle	-6	-20	-0.07
Position	0.0055	0	0.005
Heading	1.0	0	0.002

1) Body angle controller: The main objective of the inner loop controller is to drive the bot towards its equilibrium state. This controller achieves this by trying to achieve zero difference between the reference body

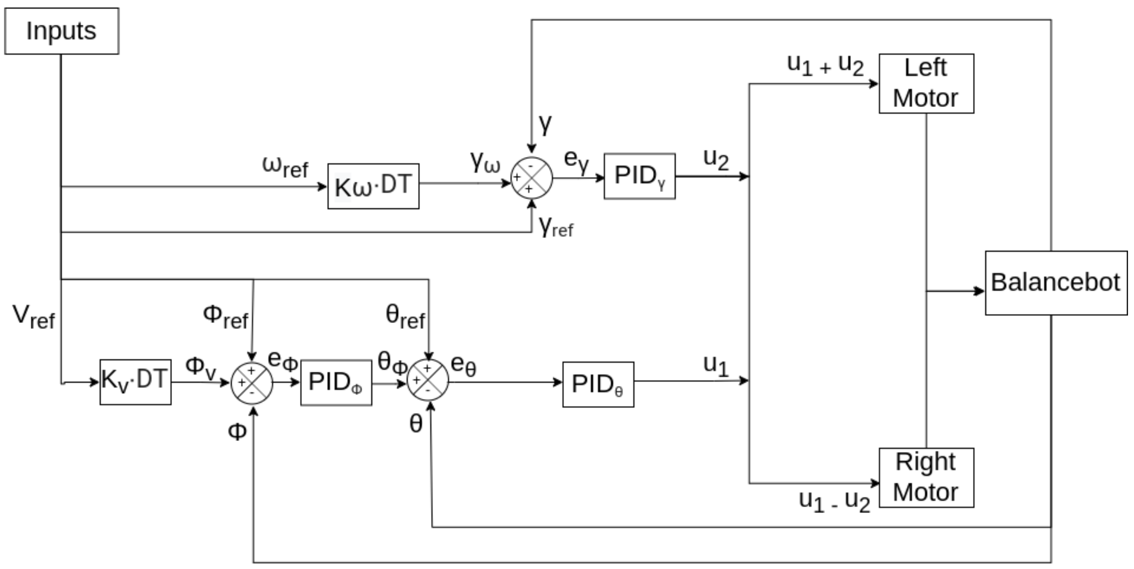

Fig. 2. Balancebot controller flow chart. There are two parallel control loops: position and orientation. The inputs of the system are reference forward velocity v_{ref}, turning velocity ω_{ref}, position ϕ_{ref}, body angle θ_{ref}. Forward and turning velocities come from DSM remote controller, and amplified by K_v and K_ω respectively. DT is sample time interval. We multiply the desired forward and turn velocity with DT, and get ϕ_v and γ_ω respectively. Use error $e_\gamma = \gamma_{ref} + \gamma_\omega - \gamma$, where γ is the current orientation from encoder feedback, to march the turning controller PID$_\gamma$, we get motor PWM command u_2. Use error $e_\phi = \phi_{ref} + \phi_\omega - \phi$, where ϕ is the current position from encoder feedback, to march the position controller PID$_\phi$, we get θ_ϕ for inner loop body angle controller. Use error $e_\theta = \theta_{ref} + \theta_\phi - \theta$, where θ is the current body angle from IMU feedback, to march the body angle controller PID$_\theta$, we get motor PWM command u_1. Finally, apply u_1, u_2 onto left and right motors, with u_2 having different signs on left/right for differential driving turning.

angle, θ_{ref}, and the body angle measured by the onboard IMU, θ by supplying counter torque to the motor drivers.

We use a discrete PID controller to control the duty cycle being applied to the motors with the values K_P, K_I and K_D as $-6, -20$ and -0.07 respectively. We obtain the following continuous time transfer function for the controller using time constant, T_f as $0.00556 s^{-1}$,

$$D_1(s) = K_P + K_I \cdot \frac{1}{s} + K_D \cdot \frac{s}{T_f \cdot s + 1} \tag{1}$$

We then obtain the following discrete time transfer function,

$$D_1(z) = \frac{-12.73z^2 + 19.48z - 6.942}{z^2 - 1.053z + 0.05263} \tag{2}$$

In addition, the motor command to body angle transfer function is characterised as,

$$G_1(s) = \frac{-331.4s}{s^3 + 170.7s^2 - 58.7s - 1633} \tag{3}$$

2) Position controller: The outer loop controller controls the wheel position of the balance bot. It takes in the error between reference wheel position ϕ_r and actual wheel position ϕ and outputs the reference body angle θ_r for the inner loop controller.

First, the position of a wheel with respect to the ground, ϕ_{ground}, is computed from the encoder value as given in equation (4). $encoder(m)$ denotes the current encoder value on wheel, m; $polarity(m)$ conforms the result to polarity convention; $gearbox$ is the gear box ratio; $resolution$ is encoder resolution.

$$\phi_{ground}(m) = \frac{encoder(m) \cdot 2\pi}{polarity(m) \cdot gearbox \cdot resolution} \tag{4}$$

Then, we average ϕ_{ground} between left and right wheel, and add the body angle θ to obtain the wheel position with respect to the body, ϕ, which is shown in (5).

$$\phi(k) = \frac{\phi_{ground}(left) + \phi_{ground}(right)}{2} + \theta(k) \tag{5}$$

We control the reference wheel position by setting the forward velocity V_f at every time step, k, and then update ϕ_r as.

$$\phi_r(k) = \phi_r(k-1) + V_f(k)\Delta t \tag{6}$$

To acquire the control signal for the inner loop, the reference body angle θ_r, we use a discrete PD controller as demonstrated in (7), where K_P and K_D are respectively proportional and derivative gain.

$$\begin{cases}\theta_r(k) = K_P(\phi_r(k) - \phi(k)) + K_D(V_f(k) - \dot{\phi}(k)) \\ \dot{\phi}(k) = \frac{\phi(k)-\phi(k-1)}{\Delta t}\end{cases} \tag{7}$$

The K_P and K_D we use are respectively 0.0055 and 0.005, and the corresponding continuous time transfer function of the controller with time constant T_f as $0.167s^{-1}$ is

$$D_1(s) = K_P + K_D \cdot \frac{s}{T_f \cdot s + 1} \tag{8}$$

We then obtain the following discrete time transfer function for the controller.

$$D_2(z) = \frac{0.03463z - 0.03431}{z - 0.9417} \tag{9}$$

The transfer function from body angle to wheel position in outer loop plant is shown in the following equation,

$$G_2(s) = \frac{-22.38s^2 + 223.8}{s^2} \tag{10}$$

3) Heading controller: We make use of the balancebots differential drive to implement the heading controller. In particular, we apply opposite duty cycles to the two motors which effects in equal torque about the bots center resulting in a change in orientation.

The current orientation is calculated as follows,

$$\gamma = (\phi_{ground}(right) - \phi_{ground}(left)) \times \frac{wheel\_radius}{wheel\_base}$$

Furthermore, the desired orientation, γ_{ref}, is determined at each time step with respect to the turn velocity, V_t, provided by the DSM transmitter as,

$$\gamma_{ref}(k) = \gamma_{ref}(k-1) + V_t(k)\Delta t \tag{11}$$

Then we apply the $\gamma_{ref} - \gamma$ as the feedback error into heading controller to get u_2. We apply u_2 with different signs onto left and right motor to obtain the change in orientation. The K_P and K_D used in heading controller are 1.0 and 0.002 respectively.

C. Odometry

We use wheel encoders and gyroscope to estimate balancebot position (x_k, y_k) and orientation γ_k in world frame at timestamp k. The odometry update algorithm is summarized in Algorithm 1 [2]. Here ϕ_L, ϕ_R are left wheel and right wheel angular position. $left\_encoder_k, right\_encoder_k$ are encoder ticks from timestamp k-1 to k. $\Delta d, \Delta\gamma$ are distance traveled and orientation change from timestamp k-1 to k. $\Delta\gamma_{gyro}$ is the orientation change detected by gyroscope. If the change in orientation difference between gyroscope and encoder is greater than $\Delta\gamma_{thres}$, then we use $\Delta\gamma_{gyro}$ for update. We encounter this situation when the balancebot runs into a bump or slip on the ground, causing large change in orientation which the encoder is unable to capture. Other constant values used in the algorithm will be discussed later in this section.

Algorithm 1: Odometry update at timestamp k

```
Result: $x_k, y_k, \gamma_k$
$x_0 = 0, y_0 = 0, \gamma_0 = 0$;
while do
    $\phi_L = \frac{left\_encoder_k \cdot 2\pi}{left\_polarity \cdot gear\_ratio \cdot resolution}$;
    $\phi_R = \frac{right\_encoder_k \cdot 2\pi}{right\_polarity \cdot gear\_ratio \cdot resolution}$;
    $\Delta d = \frac{(\phi_R + \phi_L) r_w}{2}$;
    $\Delta\gamma = \frac{(\phi_R - \phi_L) r_w}{b_w}$;
    if $|\Delta\gamma - \Delta\gamma_{gyro}| > \Delta\gamma_{thres}$ then
        $\Delta\gamma = \Delta\gamma_{gyro}$;
    end
    $x_k = x_{k-1} + \Delta d \cos(\gamma_{k-1} + \Delta\gamma/2)$;
    $y_k = y_{k-1} + \Delta d \sin(\gamma_{k-1} + \Delta\gamma/2)$;
    $\gamma_k = \gamma_{k-1} + \Delta\gamma$;
end
```

The constant parameters used in odometry model are shown in table II. The left and right polarities are the same as the motor polarities. The gear ratio and resolution are found in motor specification [1]. The wheel base is measured using digital caliper with 0.1mm uncertainty. The actual wheel base measurement is 0.204 ± 0.001m, and we use UMBark method [3] to tune the wheel base b_w value to 0.18m, to turn exactly $\pi/2$ radians as commanded.

To determine the gyrodometry threshold $\Delta\gamma_{thres}$, we measured the drift of gyroscope for 1min, in 5 trials, and found the drift rates are 0.9 ± 0.2 rad/min (mean, sd, n=5). Thus, we chose 0.09 rad, since we have about half of the drift rate of [2], we use half of the threshold value.

TABLE II
ODOMETRY MODEL PARAMETERS

Name	Value
left_polarity	-1
right_polarity	1
gear_ratio	20.4
resolution	48
wheel radius r_w	0.042 ± 0.001 m
wheel base b_w	0.18 m
$\Delta\gamma_{thres}$	0.09 rad

The verification of odometry model using drive square test is in section III-C.

D. Trajectory follower

We have two specific trajectory following controllers to perform in drag race and drive square tasks. Based on the given points we need to reach, we generate a velocity profile to feed into our controller.

1) Drag race: In drag race, our task is to race 11 meters autonomously as fast as possible. We think the best way to do this is keeping a constant body angle, which requires a constant acceleration. We generate the constant acceleration profile based on odometry distance measurement in equation 12.

$$\begin{cases} v = a\sqrt{x} + V_0 \\ b(L-x)^{1/2} \end{cases} \tag{12}$$

Here a, b are the acceleration and deceleration values, V_0 is the initial speed that the balancebot starts with, and L is the total distance. To make the velocity profile continuous, and reduce jerkiness of the balancebot, a, b are calculated by:

$$a = \frac{V_{max} - V_0}{\sqrt{\hat{x}}}$$
$$b = \frac{V_{max}}{L - \sqrt{\hat{x}}}$$

The relevant parameters are documented in table III.

TABLE III
DRAG RACE TRAJECTORY FOLLOWER PARAMETERS

Name	Value
V_{max} (m/s)	27
$\hat{x}$ (m)	9
V_0 (m/s)	10

The maximum speed V_{max}, defined as the critical velocity from which the system can recover. On trying to drive the balancebot to velocities over V_{max} would result in the bot to tip over. We select suitable acceleration distance $\hat{x}$ and initial speed V_0 by trial and error over various iterations of the drag race with different parameters.

2) Drive square: In drive square task, our strategy is to drive straight with constant forward speed v_1 along one side until it covers a distance L. Then make a counterclockwise 90^o turn with constant turn velocity v_t and forward velocity v_2. Then we reset odometry x,y values to 0, orientation to $\gamma - \pi/2$. We acquire the parameters by observing the performance of the bot on the drive square task, shown in table IV. The final trajectory is shown in figure 7 in the result section.

Algorithm 2: Drive square algorithm

```
while count is smaller than 16 do
    if $x < L$
        forward_velocity = $v_1$
    else if $\gamma - \pi/2 > \gamma_{thres}$
        forward_velocity = $v_2$
        turn_velocity = $v_t$
    else
        forward_velocity = 0
        turn_velocity = 0
        x = 0
        $\gamma = \gamma - \pi/2$
    count = count + 1
```

TABLE IV
DRIVE SQUARE TRAJECTORY FOLLOWER PARAMETERS

Name	Value
v_1 (m/s)	2.5
v_2 (m/s)	0.8
v_t (rad/s)	1.0
L (m)	0.8

III. RESULTS AND DISCUSSION

A. Parameters

We summarized the motor parameters in table V.

TABLE V
MOTOR PARAMETERS

Name	right motor	left motor
Resistance (Ω)	10.0	7.0
No load speed (rad/s)	45.5672	45.2849
No load current (A)	0.1398	0.1187
Stall torque (N$\cdot$ m)	1.2000	1.7143
Motor constant	0.0114	0.1187
Shaft friction	0.0007	0.0006
Shaft Inertia (kg $\cdot$ m^2)	4.4449e-05	3.3589e-05

The moment of inertia about three principle axes of the balancebot is summarized in table VI. We measured the period three times and use the mean to calculate the moment of inertia for each axis.

TABLE VI
MASS AND MOMENT OF INERTIA (MEAN, SD, N=3)

Name	Value
body mass w/o wheels (kg)	1.0975 ± 0.0001
J_{xx} (kg $\cdot$ m^2)	0.006 ± 0.001
J_{yy} (kg $\cdot$ m^2)	0.0086 ± 0.0005
J_{zz} (kg $\cdot$ m^2)	0.004 ± 0.001

B. *Controller performance*

To evaluate the performance of our PID controllers, we tested them on step responses, and compare the state measurements from balancebot with MATLAB simulation.

1) Body angle controller performance: We give a change of 0.1 and 0.2 rad on reference body angle, and get the resulted step response shown in figure 3 and figure 4. The body angle controller MATLAB model has rise time: 0.22s, settling time: 0.74s and overshoot: 2.69%. From the step response graphs, the controller on the real system has similar rise time, settling time. However, our balancebot isn't stable enough. The oscillation after the step caused a huge error band, which is about 0.1rad for 0.1rad step response.

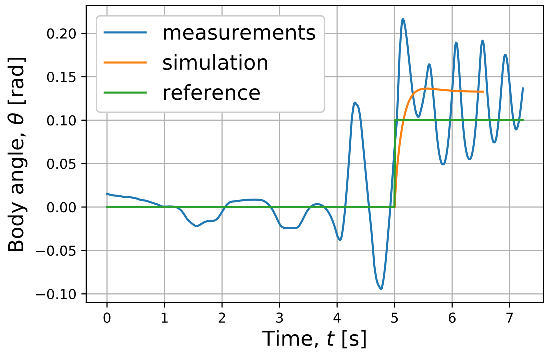

Fig. 3. Body angle controller 0.1rad step response. The blue line is IMU measurement, orange line is MATLAB model using transfer function described in II-B1, and green line is the reference body angle.

Also, the balancebot keeps accelerating to maintain the fixed non-zero body angle during the step response. When velocity becomes large, it loses control. For the case of 0.2rad step response, the balancebot fails to stabilize and tips over. 0.2 rad is the largest reference angle that we can handle, osciallating down towards settling before tipping over.

In figure 5, we give an oscillating step switching between 1.5s zero body angle, and 0.5s body angle with 0.05rad. The average measured body angle can reach the desired value, but again the oscillation is large compared to the reference body angle.

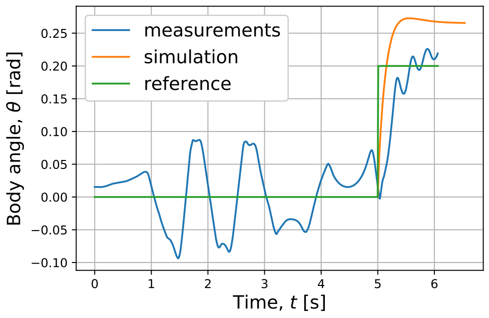

Fig. 4. Body angle controller 0.2rad step response. The blue line is IMU measurement, orange line is MATLAB model using transfer function described in II-B1, and green line is the reference body angle.

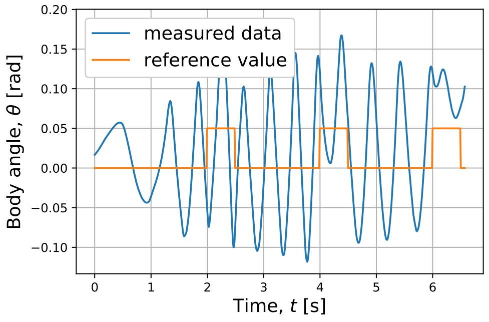

Fig. 5. Body angle controller 0.1rad oscillating step response. The blue line is IMU measurement, and green line is the reference body angle.

2) Position controller performance: We give a change of 0.2m and 0.3m on reference position, and get the resulting step response as shown in figure 6 and figure 8. The position controller MATLAB model has rise time: 0.10s, settling time: 3.71s and overshoot: 115.27%. From the step response graphs, the controller on the real system has similar rise time, settling time, and much less overshoot. The error band is about 0.1m. Also there's about 0.05m steady state error of the system.

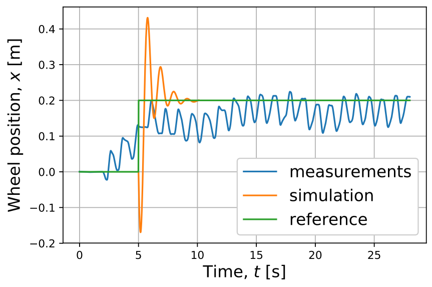

Fig. 6. Position controller 0.2m step response. The blue line is wheel position calculated from encoders, orange line is MATLAB model using transfer function described in II-B2, and green line is the reference position.

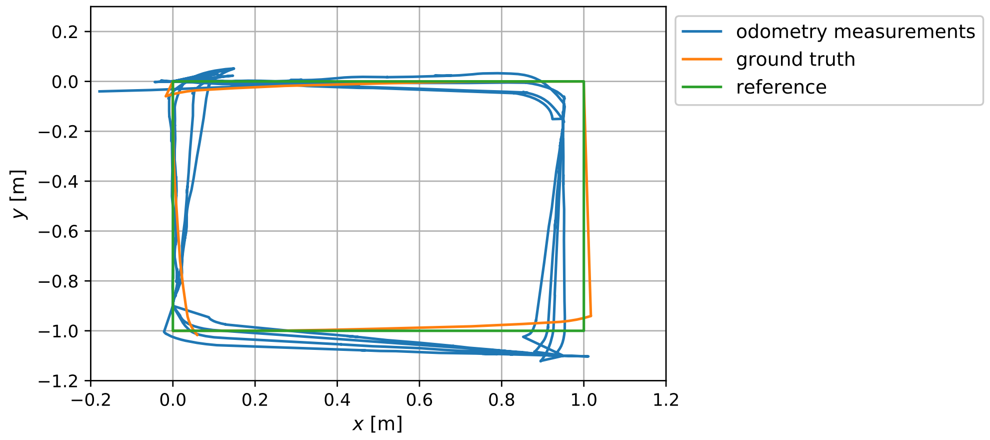

Fig. 7. Drive square trajectory. Blue lines are 4 rounds of odometry measurements on drive square. Orange line is the ground truth odometry measurement drive by hand on the same square. The green line the reference value 1m square.

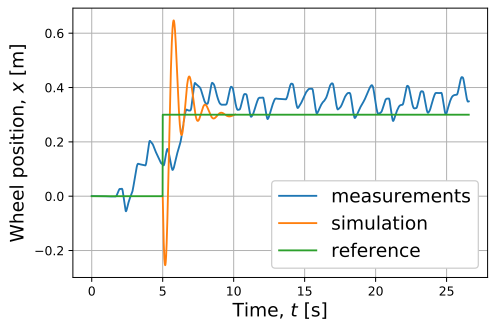

Fig. 8. Position controller 0.3m step response. The blue line is wheel position calculated from encoders, orange line is MATLAB model using transfer function described in II-B2, and green line is the reference position.

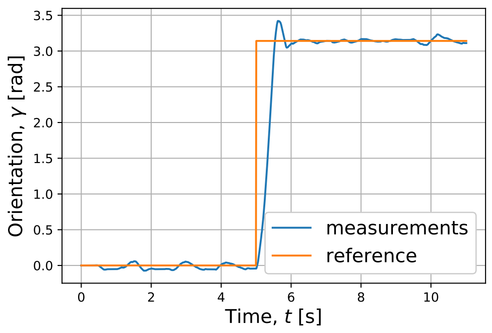

Fig. 9. Turning controller π [rad] step response. The blue line is the orientation calculated by odometry,and green line is the reference orientation.

3) Heading controller: We tested heading controller by giving it a π rad step response in figure 9, and from the plot, we can see the settling time is about 0.5s with 20% overshoot. The error band is less than 2%. Thus, our heading controller has fast and stable performance with acceptable overshoot.

C. Drive square trajectory

In figure 7, we plot the odometry measurement on drive square trajectory follower and manually push balancebot around the square. Comparing the manually drive with the reference 1m square, we can see the nominal wheel base r_w works. Also, the balancebot goes straight for 1m, so the difference of wheel diameter for left and right wheel is negligible ($\pm$0.004m).

D. Competition Performance

1) Task: Balancebot on the spot: This task required the balancebot to retain equilibrium without drifting outside a constrained 10cm region of its starting position. In addition, the balancebot was tested to check its ability to recover from disturbances introduced to it in its body angle as well as its position. The designed PID controllers were successful in maintaining its stable position for an extended period as well as recover from various perturbations to its state. However, we observed that the bot had high overshoot as well a longer settling time when subjected to a strong disturbances in its position which suggests a need for higher derivative gain, K_D.

2) Task: Drag Race: The objective of this task was to race the balancebot down a 11m track in the fastest time possible and come to stop while regaining balance within 1m after the end point. Our generated velocity profile enabled the bot to complete this task in 17.12s. During the execution of the task we notice the bot slowed down to regain body angle balance after attaining the maximum velocity. This suggests a need to futher tune the outer loop paraemeters for a faster response on wheel position.

3) Task: Drive Square: This task required the balancebot to navigate 4×4 edges of 1m square. Our odometry model was successful in completing the first 2×4 edges with minimum drift in orientation and position. For the later half of the task, we observe drift being introduced to the model additively. However, even though the bot crossed the edges on the last 3 edges, we still were able to complete the task within a drift range of $\pm 0.07m$ of the first square.

4) Task: Race Track: This task required a user to manually navigate through a set of checked in two different setting, with and without obstacles. Similar to the drag race our navigation speed by was constrained by our slow outer loop response. We also observed a need to damp our heading response at high turn velocity, V_t. In addition, abrupt change in linear velocity, V_f, introduced high frequency oscillations which pushed the system to instability.

IV. Conclusion

In conclusion, we implemented a cascaded PID controller to implement a self-balancing robot capable of navigating through complex environments. The design of the controllers required implementation of core controls concepts along with an odometry model to help the robot localize itself. We successfully completed the various challenges posed by the different tasks in the competition and identified avenues of future improvement.

References

[1] "Rob 550 balancebot motor modelling lecture rob 550 balancebot."

[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, April 1996, pp. 423–428 vol.1.

[3] J. Borenstein and Liqiang Feng, "Measurement and correction of systematic odometry errors in mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 6, pp. 869–880, Dec 1996.

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