Real-time Motion Control of Two-wheeled Balancing Robot using Cascaded PID Controller

Cong Fu, Kai Ho Edgar Cheung, Abhishek Venkataraman
{congfu, khcheung, abhven}@umich.edu
University of Michigan, Ann Arbor

***Abstract*—An inherently unstable system can be stabilized using an on board control system. In this report, we propose a methodology to keep a two-wheeled robotic system balanced while maintaining its ability to rotate and traverse. We discuss in detail the control loops and the sensory inputs that are used to keep the robot stable. Finally, we compare the odometry/IMU based trajectory traced by the robot (with and without balancing) to the ground truth available from OptiTrack motion capture system.**

Keywords- self-balancing robot, IMU based control, Odometry, Cascaded Control

I. Introduction

Self-balancing robots have been a popular robotics problem for more than a decade. It is analogous to the inverted pendulum system which has always been favorite example in physics, dynamics and control systems, etc. The most appealing part of the problem is the ease of practical realization using just a few components.

With the availability of inexpensive and reliable microcontrollers, processors, inertial motion units, many researches in inverted pendulum has converted into commercial success. One such example is the hover board, which is used as a affordable personal commute for small distances. In this project, we focus on designing the control system for balancing a two wheeled robot. We then implement additional control loops for maintaining position and heading. Finally, we test the performance of our system by traversing a pre-defined trajectory and comparing our system measurements with a highly accurate OptiTrack Motion Capture system.

II. Methodology

The harmonic integration of both softwares and hardwares are vitally important for the practical implementation of a self-balancing robot. In this section, we will first discuss about how to use the odometry model to convert encoder data into robot pose in terms of x-y position and heading in global frame. Then, we will propose our cascaded PID control structure. Finally, the discussion of the remaining supporting components would be included in subsequent sections.

A. Differential Wheel Drive

Motion control of this robot is achieved using differential drive. A differential drive consists of two independently controlled wheels, namely, left and right wheels. The linear and angular velocities of the robots can be derived from the individual wheel velocities,(ω_r, ω_l), using Equations 1, 2, 3.

$$\dot{x} = \frac{R}{2}(\omega_r + \omega_l)\cos\theta \tag{1}$$

$$\dot{y} = \frac{R}{2}(\omega_r + \omega_l)\sin\theta \tag{2}$$

$$\dot{\theta} = \frac{R}{L}(\omega_r - \omega_l) \tag{3}$$

where R is the radius of the wheel and L is the wheel base.

The inverse of the above problem is to calculate angular velocities of the wheels, (ω_r, ω_l), for a desired linear and angular velocity of the robot, $\{\dot{x}, \dot{y}, \dot{\theta}\}$. Equations 4, 5, 6 reduce the system of equations to a 2 variable problem by applying the differential drive constraint[1].

$$\dot{x} = v\cos\theta \tag{4}$$

$$\dot{y} = v\sin\theta \tag{5}$$

$$\dot{\theta} = \omega \tag{6}$$

From the above equations, we get 2 unknown variables, linear velocity, v and angular velocity, ω. We can calculate the required angular velocities for the two wheels using Equations 7, 8

$$\omega_r = \frac{2v + \omega l}{2R} \tag{7}$$

$$\omega_l = \frac{2v - \omega l}{2R} \tag{8}$$

B. Odometry

Each of the wheels has an encoder attached, which can measure the angle of the wheel. The encoder value is a signed counter and the pulse count is used to determine the angle. More details about the encoder measurement is discussed in Section II-D2. Once we know the angle of each of the wheels, we can derive the lateral and angular position as discussed below.

[1]Constraint: The wheels cannot move in a direction perpendicular to the direction of rotation.

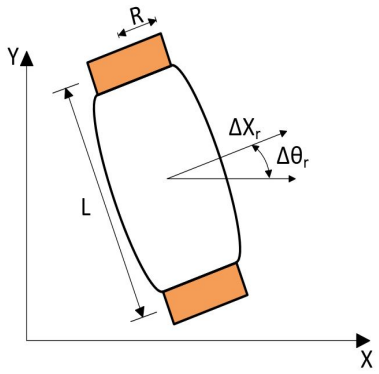

Fig. 1: Odometry model schematic 1

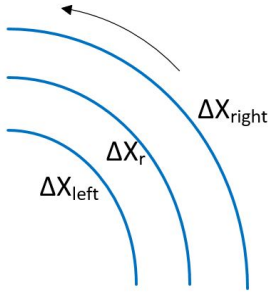

Fig. 2: Odometry model schematic 2

For any time interval Δt, the displacement of two wheels during the interval can be calculated using Equations (9) and (10).

$$\Delta d_{right} = 2\pi R \frac{\Delta \theta_l}{N} \tag{9}$$

$$\Delta d_{left} = 2\pi R \frac{\Delta \theta_r}{N} \tag{10}$$

where N is the number of total pulses per revolution and $\Delta\theta_l, \Delta\theta_r$ are the number of pulses counted by left and right encoders respectively. The change in angular and linear position of the robot can be given by:

$$\Delta\theta_R = \frac{\Delta d_{right} - \Delta d_{left}}{l} \tag{11}$$

$$\Delta d_R = \frac{\Delta d_{right} + \Delta d_{left}}{2} \tag{12}$$

The above computed linear and angular change, $\Delta\theta_R$, Δd_R can be transformed to Cartesian coordinate system as below:

$$\Delta x_w = \Delta d_R \cos \Delta\theta_R \tag{13}$$

$$\Delta y_w = \Delta d_R \sin \Delta\theta_R \tag{14}$$

$$\Delta\theta_w = \Delta\theta_R \tag{15}$$

Finally, we can get the real time position and orientation of the robot in world frame by incrementing Δx_w, Δy_w and $\Delta\theta_w$ at to the values at the previous time step.

$$\begin{bmatrix} x_w \\ y_w \\ \theta_w \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} x_w \\ y_w \\ \theta_w \end{bmatrix}_t + \begin{bmatrix} \Delta x_w \\ \Delta y_w \\ \Delta\theta_w \end{bmatrix}_{\Delta t}$$

C. Cascaded PID Controller Design

The motion of a two-wheeled robot can be defined as the ability to balance, rotate and traverse. The ultimate goal is to control such motion of a self-balancing robot to perform a path following task. The path is a 1m x 1m square given by the way-points in global frame from ground station. In this project, the given way-points would be a set of global poses including x-y coordinates and headings. All of which correspond to the corners of a square or multiple squares for more loops.

Exploiting the simple trajectory geometry, we can simplify the control strategy, and divide it into two parts.

1) Rotate to reference heading angle while balancing.
2) Drive straightly forward for 1m while balancing and heading regulating.

The partition of heading control from the overall pose control enables a heading PID controller to work in parallel. With heading controller regulating actively at the reference set-point, the translation controller would only be responsible for traveling 1m forward using cumulative distance feedback in body fixed frame from the odometry data. In addition to rotation and translation control, the balancing control should be activated in the background to ensure attitude stability while performing any rotation and translation motion.

In order to achieve the above control strategy, we devises a cascaded translation-balancing PID loop and a parallel heading control loop as shown in Figure 3. The overall motion controller of our two-wheeled balancing robot comprises three main controllers, namely the distance-velocity, heading and balancing controllers. Due to the fast dynamic nature of any inverted pendulum(ie. two-wheeled robot) , it is reasonable to separate the balancing part into a faster inner loop within the overall control structure. On the other hand, the distance-velocity controller drives the set-point of the inner balancing loop in a relatively slower manner, which enables the inner loop response to converge before executing the next set-point.

In this section. we will first discuss about how to approximate a continuous-time domain PID controller in discrete-time domain. Then, the design structure of the cascaded PID controller would be discussed in detail.

1) Discretization for real-time implementation:
In continuous-time domain, a basic PID control input, u, can be written as:

$$u = P + I + D \tag{16}$$

$$P = K_p \cdot e(t) \tag{17}$$

$$I = K_p \cdot \frac{1}{T_i} \int^t e(\tau) d\tau \tag{18}$$

$$D = K_p \cdot T_d \frac{de(t)}{dt} \tag{19}$$

Given a simpling time h, all three parts of the controller can be discretized using backward difference approximation [2]. Since the proportional part does not involve any derivative or integral term, the discretized version is simply the same as

in the continuous time domain. Therefore, the P part can be represented as:

$$P(t) = K_p \cdot (y_{ref} - y_t) \tag{20}$$

For the integral term, it can be approximated using backward difference:

$$I(t) = \frac{K_p}{T_i}(y_{ref} - y_t) \cdot h + \frac{K_p}{T_i} \cdot \sum_{\tau=0}^{t-h}(y_{ref} - y_\tau) \cdot h \tag{21}$$

which can be reformulated in recursive form:

$$I(t) = I(t-h) + K_i(y_{ref} - y_t) \tag{22}$$

$$K_i = \frac{K_p \cdot h}{T_i} \tag{23}$$

Discretization of the derivative part requires extra attention. It is because the direct implementation of the derivative term would amplify the measurement noise. Therefore, a low pass filter has to be used. First, let's consider the Laplace transformed derivative term from Equation 18:

$$D_{laplace}(s) = K_p \cdot sT_d \cdot (-Y) \tag{24}$$

The sT_d term from Equation 24 can be approximated as follows:

$$sT_d \sim \frac{sT_d}{1 + sT_d/N} \tag{25}$$

where N refers to the gain limit at high frequency. In this project, N is chosen to be 150. Combining Equation 25 and 24, the time domain representation becomes:

$$\frac{T_d}{N}\frac{dD(t)}{dt} + D(t) = -K_p T_d \frac{dy}{dt} \tag{26}$$

Again, with backward difference approximation, the low-pass filtered derivative term becomes:

$$D(t) = a_d \cdot D(t-h) - b_d \cdot (y(t) - y(t-h)) \tag{27}$$

$$a_d = \frac{T_d}{T_d + Nh} \tag{28}$$

$$b_d = \frac{K_p T_d N}{T_d + Nh} \tag{29}$$

Substituting $K_d = K_p \cdot T_d$ into Equation 29, the derivative part in terms of gains would be:

$$D(t) = a_d^* \cdot D(t-h) - b_d^* \cdot (y(t) - y(t-h)) \tag{30}$$

$$a_d^* = \frac{K_d}{K_d + NhK_p} \tag{31}$$

$$b_d^* = \frac{K_d K_p N}{K_d + NhK_p} \tag{32}$$

2) Cascading multiple PID controllers:
As discussed at the beginning of this section, the overall control input consists of three components, $u_{balancing}, u_{velocity}\&u_{heading}$. Then, the control inputs for left and right wheel can be defined as follows :

$$u_{overall,R} = u_{balancing} + u_{vel} + u_{heading} \tag{33}$$

$$u_{overall,L} = u_{balancing} + u_{vel} - u_{heading} \tag{34}$$

where $u_{balancing}$ corresponds to the control input from the innermost balance PID controller. Likewise, the $u_{velocity}$ term belongs to the outer control loop which sets the pitch reference for the inner balancing loop. Noted that, the only difference between the left and right control input is the $u_{heading}$ term. Such difference in left and right wheel control signal provides the self-balancing robot the capability to rotate.

The overall cascaded PID structure is shown in Figure 3. As one can see in the figure, the cascaded part of the PID loop controls only the translation motion of the balancing robot, while the heading controller would operate in parallel.

The outer loop consists of two layers (blue-loop): a velocity and a relative distance control loop. The outermost distance controller takes in the distance set-point (1m) from the groundstation, and computes the velocity reference point for the second layer velocity controller. Both controllers operate in a sampling rate of 30Hz. An algorithmic illustration in Algorithm 1 depicts the basic structure of this loop. Define $\xi(\cdot) = P(\cdot) + I(\cdot) + D(\cdot)$:

Algorithm 1 Outer control loop

function *outer_loop*
  global variables
    ODO //contains body fixed distance and velocity
    θ_{ref}
  end global variables
  while True **do**
    $u_{dist} \leftarrow \xi(d_{ref} - ODO.dist)$
    $v_{ref} \leftarrow saturation(u_{dist}, v_{min}, v_{max})$
    $u_{vel} \leftarrow \xi(v_{ref} - ODO.vel)$
    $\theta_{ref} \leftarrow saturation(u_{vel}, \theta_{min}, \theta_{max})$
    $sleep(0.33)$ $\quad //30Hz$
function $saturation(U_{in}, S_{min}, S_{max})$
  $U_{out} \leftarrow U_{in}$
  if $U_{in} > S_{max}$ **then**
    return $U_{out} \leftarrow S_{max}$
  else if $U_{in} < S_{min}$ **then**
    return $U_{out} \leftarrow S_{min}$
  return U_{out}

The balancing controller is embedded in the inner loop (red-loop), where it runs at a faster sampling rate at 100Hz. It receives θ_{ref} from the velocity controller in the outer loop. In addition to the cascaded control loop, the heading controller operates in parallel with the balancing controller in the same sampling rate to correct or regulate the heading angle while

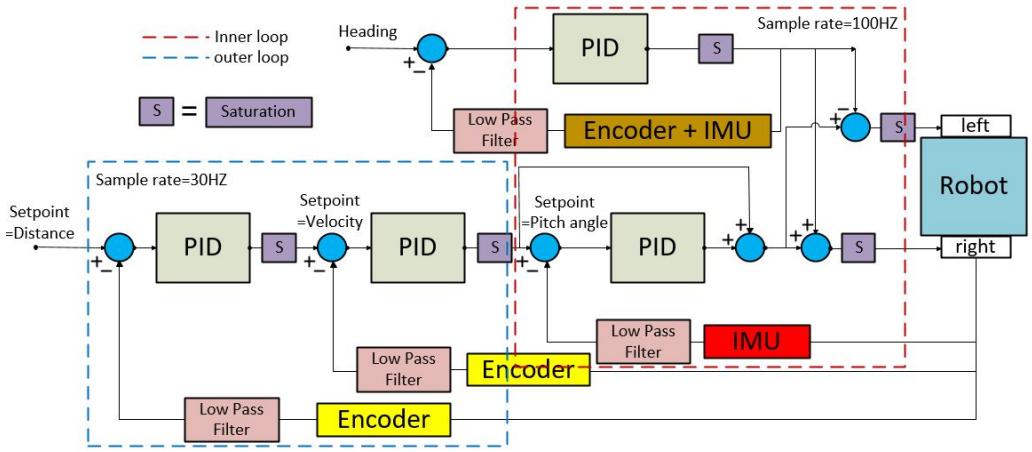

Fig. 3: Cascaded PID control loop

rotating and traversing. Similarly, the algorithmic form of the inner loop is shown below in Algorithm 2.

Algorithm 2 Inner control loop

function *inner_loop*
  global variables
    IMU //contains pitch info
    ODO
    θ_{ref}
  end global variables
  while True **do**
    $u_{balancing} \leftarrow \xi(\theta_{ref} - IMU.\theta)$
    $u_{balancing} \leftarrow saturation(u_{balancing}, u_{min}, u_{max})$
    $\psi \leftarrow sensor\_fusion(ODO.\psi, IMU.\psi)$
    $u_{heading} \leftarrow \xi(\psi_{ref} - \psi)$
    $sleep(0.01)$ $//100Hz$

Noted that sensor fusion is employed when reading heading angle from odometry and IMU data. Details of sensor fusion is discussed in the gyro integration section. Moreover, all control input saturations would be summarized in Table I.

D. Sensory inputs

A feedback loop acts on the error between the set-point and the observed output. Hence, it is essential to accurately measure the output of the system. In our system, the state is measured by acquiring date from the below listed methods

1) Inertial Motion Unit (IMU): On the chassis of our robot, we mounted MPU-9250, a 9 DOF IMU(3 axis magnetometer, 3 axis accelerometer, 3 axis gyroscope). The raw measurements are processed by on-board Digital Motion Processor™, and outputs orientation data in the form of Tait-Bryan (Euler) angles over the I2C bus. Once the data is available in the IMU, it generates an interrupt and which is fed to a GPIO pin of the BeagleBone. On receiving this interrupt, the robot state is updated with the available readings.

Our system consists of moving mechanical parts such as the motors, which create high frequency vibrations. Apart from the true orientation of the robot, the IMU also picks up these high frequency noise. In order to ensure that this noise is not fed to the control loop, a low pass filter. The output of IMU was filtered using an exponential smoothing, Equation 36. We preferred this over other low pass filters such as moving average or down-sampling, due to its implementation simplicity. We used a smoothing factor, $\alpha = 0.05$ for pitch and yaw measurements. The filtered pitch and yaw was then fed to the balance and heading control loop respectively. Figure 4 compares the raw vs filtered values when random perturbations are given to the robot.

$$s_0 = x_0 \tag{35}$$

$$s_t = \alpha x_t + (1 - \alpha) s_{t-1}, t > 0 \tag{36}$$

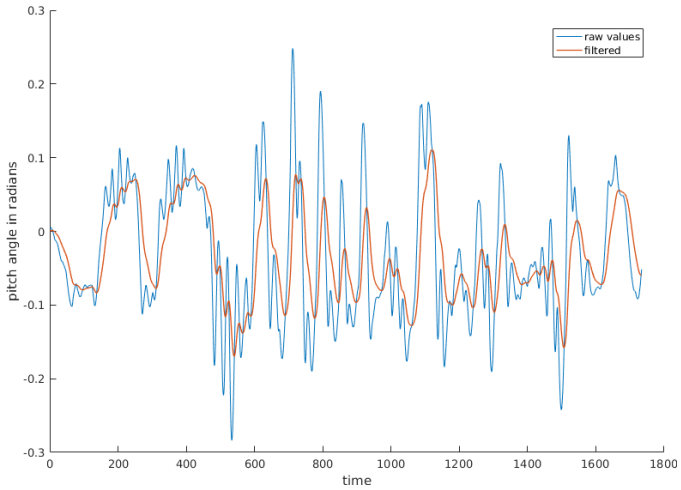

Fig. 4: Low pass filter

2) Wheel Encoders: As mentioned in Section II-B, we estimate the position of the robot using the position of the left and right wheel. In our system, there are quadrature encoders coupled to shaft of each of the left and right motor. Figure 5 visualizes the signals from a quadrature encoder. The angle of the wheel is proportional to the count of the pulses generated by the encoder. The phase difference between the A and B

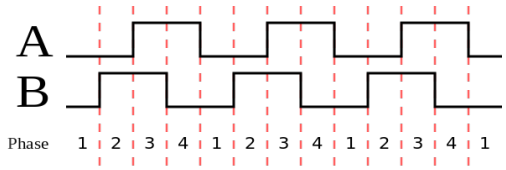

Fig. 5: Encoder signal

channels provide the direction of rotation. For every pulse from the encoder, the angle of the wheel is given by the equation 37, where P is the count encoder pulses, R is encoder resolution, G is the gear ration. In our case, G= 20.4, R= 48 pulses/revolution. The distance moved by the wheel, d_wheel can be got from the equation 38, where D_{wheel} is wheel diameter = 0.08m.

$$\theta_{wheel} = \frac{2\pi \times P}{R \times G} \tag{37}$$

$$d_{wheel} = \theta_{wheel} \times \frac{D_{wheel}}{2} \tag{38}$$

3) OptiTrack Motion Capture: In order to quantitatively evaluate our implementation, we measured the true position of the robot using OptiTrack Motion Capture system. The system consists of 6 IR cameras facing the region of interest. The object to be tracked, (the robot in this case) is affixed with 3 or more retro reflector markers. The cameras detect these markers and provide accurate pose of the object. More details of the OptiTrack system can be found at http://optitrack.com/products/prime-13/. We used 4 makers placed in a form of trapezoid and used the $\{x, y, \theta\}$ pose to compare with odometry pose.

E. Gyro Integration

Since wheels are prone to slippage, odometry-based pose estimation error would accumulate over time. One way to solve this problem is to apply sensor fusion using IMU data. A simple sensor fusion function for yaw angle or heading is shown in Algorithm 3

Algorithm 3 Sensor fusion method

```
function sensor_fusion(ODO, IMU)
    threshold ← 0.1
    φ_final ← ODO.φ
    if ABS(ODO.φ − IMU.φ) > threshold then
        φ_final ← IMU.φ
        ODO.φ ← IMU.φ
    return φ_final
```

With the cascaded PID structure defined, implementation and performance evaluation would be discussed in subsequent sections.

III. Implementation

A. Elevating center of mass

The position of center of mass (CG) is an important parameter while designing the system. In fact, the difference between a simple pendulum and an inverted pendulum is the position of its CG about the pivot point. In case of a simple pendulum, the gravitational force acting on the CG provides the restoration force to stabilize the system. However, in an inverted pendulum, the gravitational force acts in the same direction as the displacement and hence does not stabilize the system.

In the case of a self-balancing robot, angular moment of inertia of body opposes the direction of motion of wheels. This opposition to motion acts as the restoring force required to balance the robot. The forces acting on the CG are shown in Figure 6 The angular moment of inertia is proportional to the distance of CG to the pivot point. In order to keep the CG at the appropriate height, it becomes important to distribute the weight correctly.

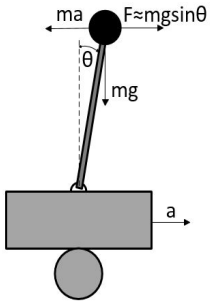

Fig. 6: Forces acting on an inverted pendulum model

B. Tuning PID

For tuning the PID controller, start with the inner most loop. After we tune and test all the parameters of the current loop, we move to the subsequent outer loop. For each of the PID loops, we first set K_i and K_d to zero and increase K_p until the robot is balancing about the set point. It is important to check that K_p is set such that the robot can recover back to the set point position. Subsequently, we increase the K_d to damp the oscillation. An observation we found during tuning K_d is that it is very sensitive to high frequency noise. Hence it is vital that the low pass filters are tuned before proceeding to this step. Finally, if we find some steady state error, we increase K_i. We found the integral part to be very useful in cases of balancing and distance. However, we had to clamp the integral component as we had problems with overshooting.

As mentioned earlier, we first tuned the balancing (inner) loop. Though robot would stay upright, it would drift away and not maintain position. To fix this, added and tuned the velocity control loop. Since we used average velocity over 50 time steps, the control loop would keep the robot at a place. However, this loop could not contain large displacements or drift over longer period of time. To circumvent this issue, we added a distance loop as the outermost control loop. We used the above described strategy for each of the control loops.

Apart from, the 3 cascaded control loop, we added a parallel control loop for maintaining heading. This output of this loop created a difference between the speeds of the left and the right motors, which would correct the heading. Table I shows the final parameters for each of of PID controllers were tuned.

TABLE I: PID Parameter Table

Loop	K_p	K_d	K_i	S_{min}	S_{max}
Distance	0.5	5	0.01	-0.05	0.05
Velocity	1	0.1	0	-0.1	0.1
Balance	8	0.12	130	-0.8	0.8
Heading	0.1	0.01	1.0	-0.8	0.8

While a basic cascaded control sounds appealing, we did not get the expected performance. This was attributed to the lag in system response. We addressed this problem by feeding forward velocity signal, as shown in Figure 7). With the feed-forward loop in place, the velocity loop apart from changing the set point of the balance angle, added an additional off-set to the wheel speed. This technique, helped increase the responsiveness of the system, this making the robot stable.

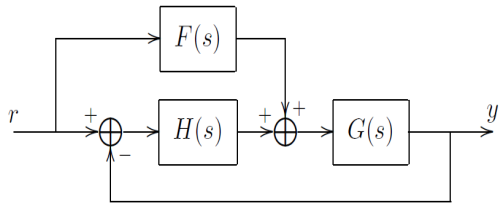

Fig. 7: Feedforward control scheme [1]

C. Operational Limits

Though the control system keep the robot stable while within the operating range, we had to add additional checks to ensure to keep the robot/system safe when it exceeded the design specification.

1) Tipping angle detection: Tipping angle is the angle beyond which the robot may be unsafe to operate. The recovery is considered unsafe because the velocity required to recover from this angle would exceed the specifications of the motor. The tipping angle for our robot was found to be ± 0.35 radians. Whenever the pitch (in radians) of the robot was outside $\left[-0.35, +0.35\right]$, the system would turned off the motors.

2) PWM Clamp: Though there is individual saturation limits on both the heading and balance pwm, as mentioned in Table I, under certain situation the sum can exceed the range $\left[-1, +1\right]$. When the PWM value of the motor exceeds the valid region, it could lead to undesired results. The clamp function ensures that the PWM are scaled such that the values are within limit and the intended action is executed. Algorithm 4 explains the logic of the clamp function.

Algorithm 4 PWM clamp function

function $PWM\_clamp(PWM_{left}, PWM_{Right})$
  if $abs(PWM_{left}) > abs(PWM_{Right})$ **then**
    if $abs(PWM_{left}) > 1$ **then**
      $PWM_{Right} = PWM_{Right}/abs(PWM_{Left})$
      $PWM_{Left} = copysign(1, PWM_{Left})$
    else if $abs(PWM_{Right}) > 1$ **then**
      $PWM_{Left} = PWM_{Left}/abs(PWM_{Right})$
      $PWM_{Right} = copysign(1, PWM_{Right})$
  return PWM_{left}, PWM_{Right}

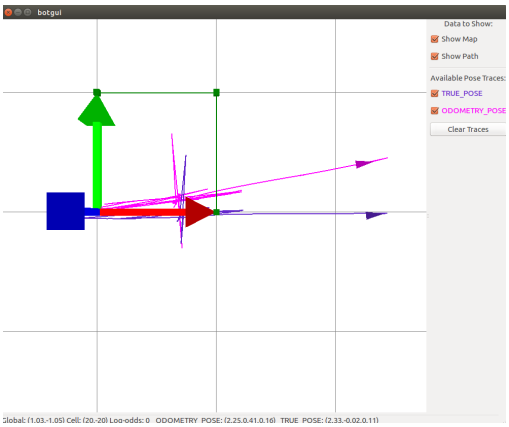

Fig. 8: Driving around in RC mode

IV. Results and Discussion

A. Integrating Motion Capture with BotGUI

After the OptiTrack system was integrated, we were able to plot both the true position and odometry-based position estimation. Figure 8 shows one of the real-time streaming pose tracing scene on BOTGUI. In the subsequent drive square tasks, the measurement difference between the two becomes clearer. The error in odometry is caused by the slipping between the wheels and the floor. Since the encoders measure the shaft angle and not the actual displacement, any slippage would result in excess odomotry distance measurement. Additionally, since the current estimation of odometry is based on the previous estimation, the error would propagate over time. However, unlike odometry, the OptiTrack is based on a stationary camera system and does not accumulate error over time.[2]

B. Drive Square: Encoder based Odometry

We tested our odometry implementation by driving the robot along a square path with a side of length 1m. Figure 9 overlays the path traced by odometry-based measurement on top of the ground truth subscribed from OptiTrak. The video `caster without IMU.mov`, shows the demo of the task.

It can be seen from Figure 9 that though the robot believes that it is going in a square path, it is quite far away from the true path. Since the robot has 3 points of contact, it is easy for the wheels to slip, which causes the drift as shown in the figure. This is reinforced by the observation that while the robot is moving in a straight path, the error does not increase substantially. Most of the error is accumulated during the turns.(turns are points in the graph where the X and Y coordinates do not change substantially). In order to reduce the effect of wheel slippage, we fused the heading data from the IMU. We discuss that in the next section.

[2]This is assuming that the setup is calibrated and is not disturbed during the task.

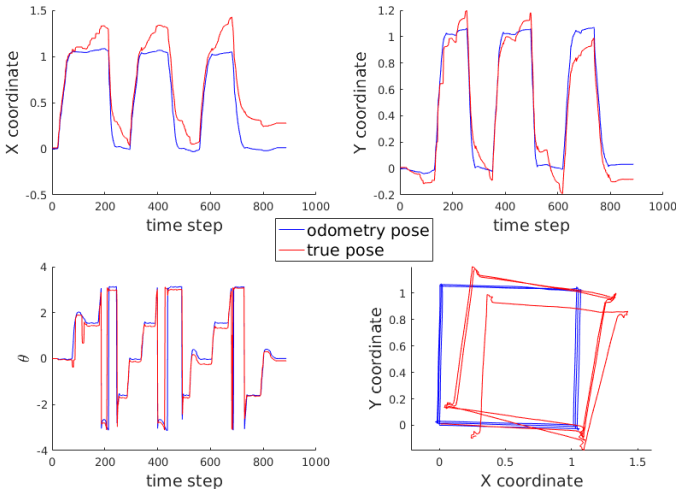

Fig. 9: Driving in a square with caster wheel using without gyro fusion

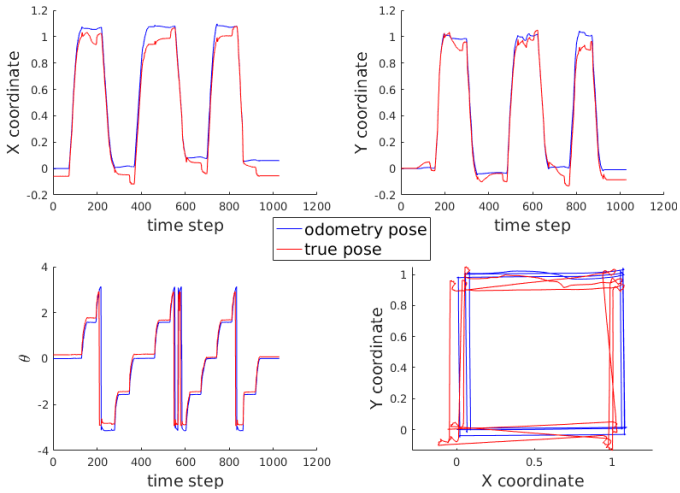

Fig. 10: Driving in a square with caster wheel using gyro fusion

C. Drive Square: Gyro + Encoder based Odometry

As shown previously, encoder-based odometry performs well in straight path, but deteriorates the performance in turns where the wheels are prone to slippage. The heading error caused during rotation motion is addressed by integrating the heading information from the IMU as mentioned in Section II-E. Figure 10 overlays the path traced by sensor measurement on top of the ground truth subscribed from OptiTrak. The video `caster with IMU.mov`, shows the demo of the task.

The improvement can be seen by comparing the Figures 10 and 9. The path traced using sensor-fused measurement is much closer to the ground truth square path than that of the pure encoder-based odometry. Figure 11 overlays and compares the errors in X and Y coordinates for the two methods. A point by point comparison would not be possible since the two runs were performed at different times. However, we can notice that the errors induced by pure encoder-based odometry propagate and diverge over time. The mean and variance of error for the complete run is shown in Table II. It is important to note that the variance of the error is significantly higher in case of pure encoder based odometry.

TABLE II: Error Mean and Variance for drive square

Error	Encoder only		Encoder+IMU	
	Mean	Variance	Mean	Variance
X	−0.1863	0.0124	0.0680	0.0015
Y	0.0404	0.0088	0.0377	0.0019
θ	0.1162	0.0793	−0.1566	0.0565

D. Balance RC Driving

The balancing loop of the robot was initially tuned without adding the drive square loop. The PID gains were tuned and the values are tabulated in Table I. After tuning the balance PID, we added the outer loop embedding the velocity and distance controllers. It ensured that the robot did not drift away in a translational sense from any reference location. To test

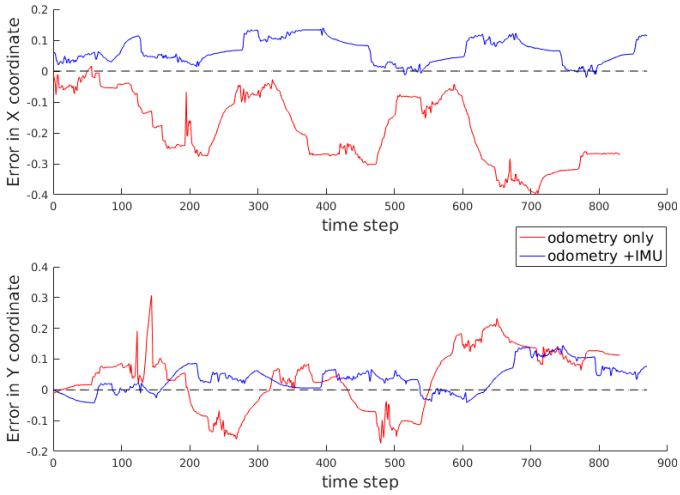

Fig. 11: Comparison of error between pure odometry and gyro+odometry

control robustness, the robot was given a perturbation and was successfully holding to the set point. Figure 12 shows that the robot returns to position after it was given some perturbation in Y direction. Additionally, we added a heading loop which sets a differential speed between the left and right wheels to maintain heading set point.

Figure 13 shows driving forward with RC while maintaining balance. The wavy curve of the odometry pose shows that the robot is balancing. However, the flat line for true pose may be because the OptiTrack system was not able to locate the robot at time.

Figure 14 shows turning with RC while maintaining balance. It can be seen that the robot can successfully maintain position and respond to movement commands while maintaining balance.

E. Drive Square: Balanced Control

After testing and integrating individual blocks described above, we successfully completed the challenge of driving

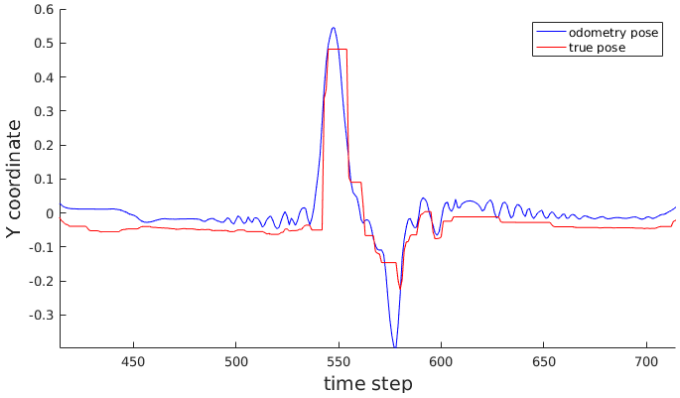

Fig. 12: Perturbation given while holding set point

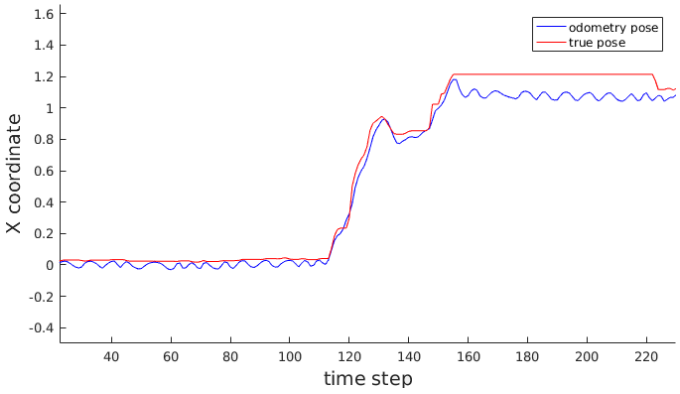

Fig. 13: Drive Forward with Radio Controller

the robot in a square while keeping it balanced. The video `balance.mov`, shows the demo of the challenge task.

The path traced by the robot overlaid with the ground truth is shown in Figure 15. It depicts that the robot is consistently tracing the path. However, the angular offset between the odometry position and the true position is due to initial gyro offset[3]. It can be seen from the plots that this error is comparable to the ones with the caster wheel. This demonstrates that the integration of the balancing loop does not affect the original performance of our robot. In the same figure, it is also interesting to note that the discrepancy between the traces from both the robot odometry and motion capture is high while the robot is maintaining a set point. The reason behind this is that the measurement of position for odometry is done with respect to the wheel, while the same measurement from OptiTrack is done with respect to the top of the robot. Therefore, the odometry measurement would appear to oscillate in a larger amplitude as compared to that of the OptiTrack measurement. During balancing, the two positions of the robot do move in the same direction since the pitch angle changes. This is shown in Figure 16.

[3]As future work, we would like to fix this error by adding an initial gyro offset at the start of drive square execution.

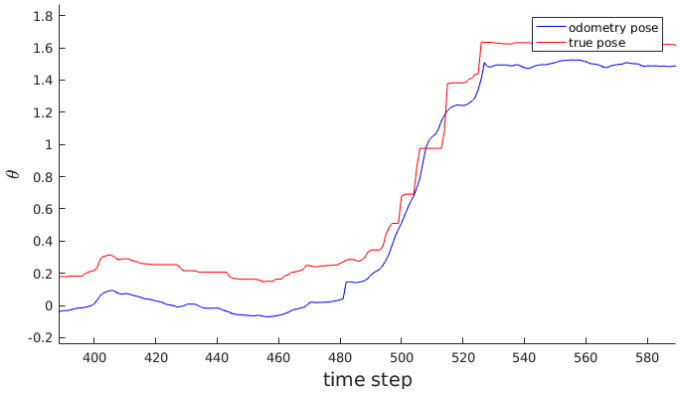

Fig. 14: Turn with Radio Controller

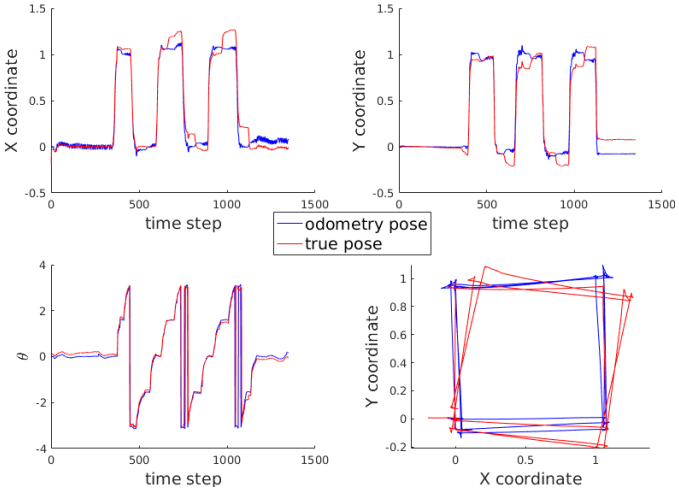

Fig. 15: Driving in a square while balancing the robot

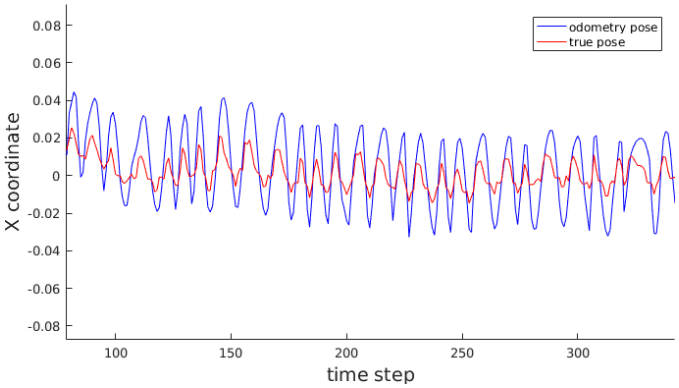

Fig. 16: Disparity between true x-position and odometry position while holding a set point

V. Conclusion

We have implemented a cascaded PID control system for a two-wheeled robot. The robot can trace a pre-defined path successfully while maintaining balance. Finally, we have compared the performance of odometry-based pose estimation to that of a highly accurate motion capture system. From the results, we can see that our system performs reasonably well.

CERTIFICATION AND PEER EVALUATION

I participated and contributed to team discussions on each problem, and I attest to the integrity of each solution. Our team met as a group on

20-Feb-17, 21-Feb-17, 21-Feb-17, 22-Feb-17, 6-Mar-17, 8-Mar-17, 11-Mar-17, 12-Mar-17, 13-Mar-17, 15-Mar-17, 18-Mar-17, 19-Mar-17, 20-Mar-17, 21-Mar-17.

Cong Fu

Kai Ho Edgar Cheung

ABHISHEK-VENKATARAMAN

ACKNOWLEDGMENT

We would like to thank Prof. Ella Atkins, Dr. Peter Gaskell, Mr. Theodore Nowak, Mr. Abhiram Krishna, our fellow students in ROB 550 course for the helping and inspiring us through out the duration of the project. We are grateful to the Robotics Institute, University of Michigan for providing us with all the resources needed for the project.

REFERENCES

[1] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. Robot modeling and control. Vol. 3. New York: wiley, 2006.
[2] Karl-Erik Arzen. Real-Time Control Systems. KFS i Lund AB, 2014

APPENDIX A

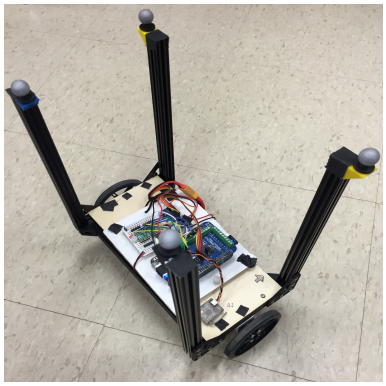

Fig. 17: Fully functioning balancebot