Gyrodometric-based Navigation System for a Two-wheeled Self-balancing Robot

Zihang Wei
Electrical Computer Engineering
University of Michigan
Ann Arbor, Michigan 48105
Email: wzih@umich.edu

Boliang Liu
Electrical Computer Engineering
University of Michigan
Ann Arbor, Michigan 48105
Email: boliang@umich.edu

Rajashree Ravi
Robotics
University of Michigan
Ann Arbor, Michigan 48105
Email: rajravi@umich.edu

***Abstract*—Two wheeled balancing robots are a good platform for testing the efficiency of various control schemes. The need to be precise and robust while navigating it through an environment is an interesting aspect of the inverted pendulum based robot model that is analysed. The main objective of this paper is to implement a gyrodometric-based dead-reckoning navigation system on a differential drive robot, and to control the pitch angle, velocity and position of the robot using a cascade control scheme and sensor feedback to maintain balance while traversing a commanded path even when disturbance is introduced in the system. Various test cases are implemented to test robot navigation and analyse resultant error. Optitrack motion tracking system provides ground truth values that are compared against the robot's gyrodometry and odometry data. The paper also introduces a propeller design as a possible additional control system that can balance the robot even beyond tipping angles.**

Keywords - Balance, Differential drive robot, Odometry, Gyrodometry, Cascade control, Path planning, State machine, LCM

I. Introduction

The purpose of the BalanceBot Lab was to introduce the concept of dynamic control of DC motors to balance a wheeled inverted pendulum, navigate it using an odometric dead-reckoning system, integrate gyroscope data from an inertial measurement unit (IMU) to increase accuracy and robustness and to implement these objectives using an embedded system. This report discusses the mechanial and electrical construction of a two wheeled differential drive robot and the use of a cascade control scheme to control balance, forward velocity, angular velocity, position and heading of the robot, the state machine used by the robot for switching between holding a set-point, RC driving and autonomous driving in a square and the use of Optitrack to analyse the accuracy and errors involved in trajectory tracking.The report is divided into the following six sections which discuss the above stated objectives:

Section II discusses the mechanical and electrical design of our BalanceBot, describing the use and placement of various components to obtain an optimally structured robot that helps in designing an efficient controller for balance. It also introduces a propeller design that can increase the stability and analyses its advantages and limitations.

Section III discusses the software design and architecture for implementing the state machine and accessing various test cases and functions.

Section IV discusses the implementation of odometry in a differential drive robot and the integration of IMU data to increase robustness to disturbances and slipping of robot wheels.

Section V discusses our cascade controller scheme, its implementation and tuning of various parameters for control of velocity, balance and trajectory tracking.

Section VI describes our methods of data acquisition using Optitrack system and the analysis of the error data obtained from comparing the ground truth values from Optitrack, to that of the robot's gyrodometry or odometry data.

Section VII presents a discussion of our design, system performance and possible future work to improve the performance.

II. Hardware Design

A. Mechanical Design

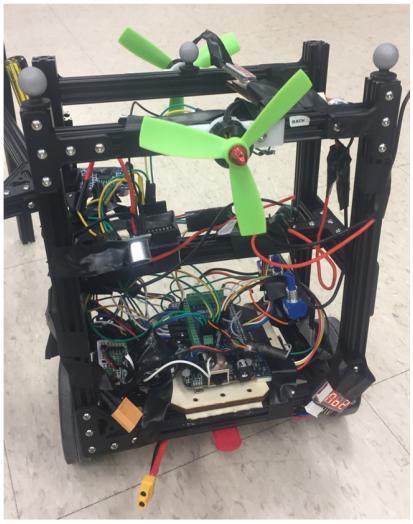

Fig. 1. Robot mechanical design.

1) Bone Structure of Frame: While designing our robot, center of gravity of the whole system was an important

consideration. Though we initially considered a lower centre of gravity to increase the inherent stability of the robot, we later realised that a higher center of gravity resulted in having to command a lower pitch angle range to control acceleration, which in turn gave a better performance from the balance controller. However, a higher center of gravity would introduce more inertia about the top of the robot, which lowers the efficiency of the balance controller, making the system more susceptible to fall over. After an analysis of the tradeoffs, we decided on the following design for the robot to lower the range of pitch angles that would need to be commanded while still accomodating for the range of values over which we could tune our parameters.

Aluminium extrusions from OpenBeam were used to build the robot frame along with a laser cut wooden board that is attached to the beam using double sided foam tape to prevent any contact with the metal frame. The tape was also used to fix the other circuits to the board and frame to avoid any short-circuiting.Custom parts were fabricated using rapid protyping techniques parts to attach the Arduino module and two propeller motors to the frame. Two 12V DC motors were attached to the frame and the wheels which were at a distance of 21.7cm from each other. A BeagleBone Black and a robocape was attached to the top of the wooden board along with an IMU and motor driver circuits. Furthermore, potentiometers were connected with the BeagleBone to tune PID values and ultrasound sensors were attached to the arduino placed on the middle beam at an angle approximately equal to 45^o. This solid design also acts as a mechanical suspension to reduce the influence of vibrations,thus enhancing the accuracy of IMU.

B. Propeller and Distance Detection System

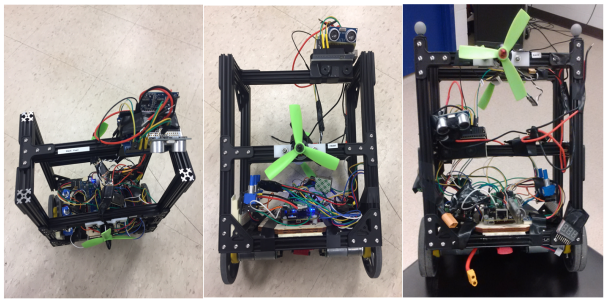

Fig. 2. Robot mechanical design.

We propose the use of propellers attached about a moment arm, at the top of the robot to have maximum momentum and torque to balance the robot when used. The design proposed is as shown in Fig. 2. One aspect to consider while using the propeller is that it needs to be constantly run at a minimum speed while upright. If the propeller starts when only the robot starts to tip, the lag time in starting the motor would already result in the robot having fallen over. However, if while trying to prevent this we choose to run the propeller constantly, it becomes difficult to tune the PID controllers of the propeller

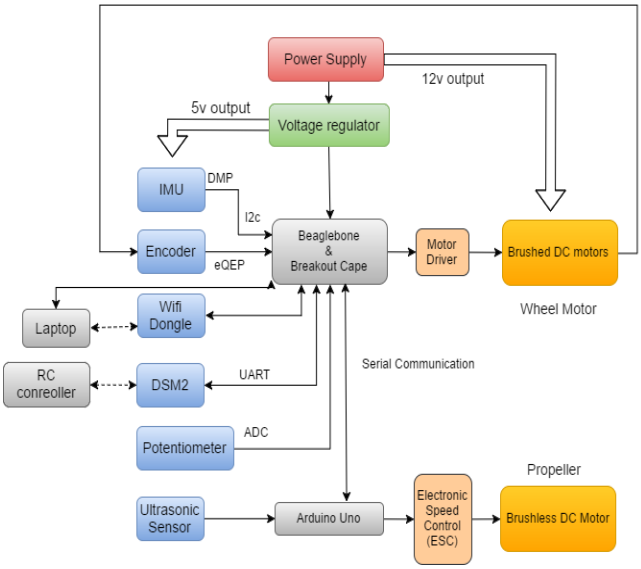

Fig. 3. Robot mechanical design.

and the wheel independently such that it would complement each other. Also, we found that if the propeller is chosen to operate only when the robot is in danger of tipping over, it requires a higher amount of thrust than is capable of the propeller currently installed in the robot, as the robot was quite heavy. Hence, even though the concept of the propeller being used to balance the robot is viable, our usage of the wheels to control the velocity and balance in a compact and efficient manner overweighs the advantages of the propeller.

C. Electric Module Design

Our robot system includes several electrical modules that are interfaced as shown in Fig. 3. The subsystems are as listed below.

`Beaglebone Black(BBB)` : Beaglebone is a low-power, open-source, single-board computer, running Debian OS. In this project, because of certain kernel patch problems, we can only use it as a near real-time control system. This near real-time system though, proves sufficient for the implementation of all desired functionalities like the cascade control scheme, and data acquisition through IMU interrupts,etc.

`Breakout Cape` : The Breakout Cape is an extension and interface of the Beaglebone board. It has screw terminals for power, twelve GPIO pins, seven ADC channels. two UARTs, one I2C bus (I2C2), one SPI bus (SPI0) and a second SPI bus that can be run in real time from the programmable real-time units (PRUs) and five 3-pin headers for hobby servos. This is used to primarily isolate and protect the BBB from potentially damaging back EMFs produced by the motors

`Motor Module` : This includes a brushed DC motor operating at 12V, a 34:1 ratio gear box, a 48 ticks/shaft rotation disc, and an incremental quadrature encoder.

`Motor Driver` : Two MAX14870 Single Brushed DC Motor Driver Carrier are used to take a low-current control signal and then turn it into a higher-current signal that can

drive the two brushed DC wheel motors.

`IMU` : MPU-9250 (IMU) contains an accelerometer, gyro and a magnetometer and has a special digital motion processor that can output fused orientation data in the form of quaternions and Tait-Bryan (Euler) angles.

`DSM2 Receiver` : This is used to communicate with the RC transmitter for RC controlled driving and is bound to a particular controller which is configured to have channels 2 and 3 for linear and angular velocities.

`Voltage Regulator Board` : A 5V/2A voltage regulator board is used to downsize the voltage from 12V(power for the motors) to 5V(power for Beaglebone board and other circuits).

`Arduino Uno` : It is an open source, real-time microprocessors board from which time-accurate data from ultrasonic sensor can be obtained to control the propeller module.

`Untrasonic Module` : HC-SR04 is an ultrasonic distance sensor module that is used to perceive the distance to the ground.

`Wifi Dongle` : We use the Wifi module to build communication channel between laptop and Beaglebone.

`ESC module` : The electronic speed control module is an electronic circuit used to vary an electric motor's speed and direction.It is used to control our propeller motors through the Arduino board.

`Potentiometer` : The potentiometer module we built is used to tune PID parameters. The voltage of potentiometers can be read by Beaglebone ADC module. This module increases the efficiency of tuning parameters as we can dynamically change the PID value without having to recompile code everytime we need to change values.

Enhanced Quadrature Encoder Pulse (eQEP) decoder module is built into the BBB's TI Sitara ARM processor. The TI eQEP hardware peripheral that is integrated into the BeagleBone processor can accurately decode quadrature encoded pulses. The unit allows one to attach a quadrature encoder to the SoC directly and have it handle the counting of the encoder ticks, versus needing an external controller such as an Arduino to do the decoding process. This module gives us high speed data from the encoder and the IMU external interrupt timer makes the time interval and frequency (200Hz) at which data is obtained very accurate.

We also define a tip angle. When the robot tilts at an angle larger than the tip angle, it will stop the motors. We have two ultrasonic sensors to detect the distance of the ground from the front and back of the robot. If the two detected distances are more than a threshold value, then the robot is considered to be upright and the motors will run at minimum required speed. The ultrasonic sensor is connected with Arduino, and Beaglebone will use serial communication to receive data from Arduino at a low baud rate (because the motor stop mode is low priority, there is no need to take a lot of system resources).

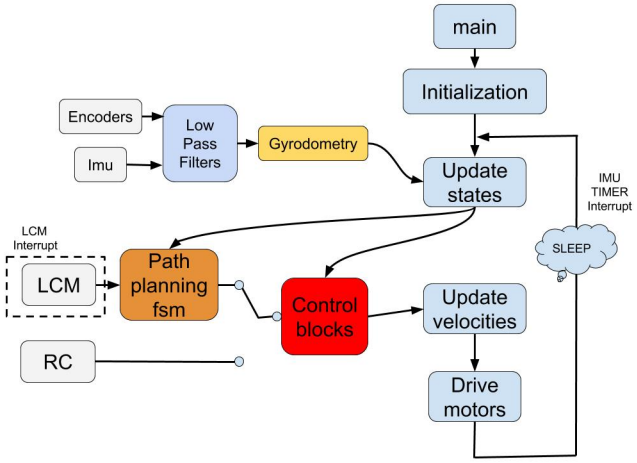

Fig. 4. Main Thread Architecture.

III. Software Design

In this project, we design the software architecture and state machine to make the process of deciding control signals an intuitive and systematic procedure.

A. Software Architecture

A multi-thread software system is implemented for the control of the BalanceBot. The main thread is designed for path and balance control and optional threads such as printing thread and lcm message publishing thread are integrated for debugging and to display in real-time the robot state.

The main event-driven thread architecture is shown in Fig. 4. After device initialization, the robot falls into sleep to reduce power consumption and waits for interrupts. Two types of interrupts can wake up the robot. One is the IMU timer interrupt, which is set at the I2C's highest communication rate between Beaglebone and IMU. Once the interrupt occurs, the program updates robot's states, including current speed, heading and pitch angles. The other is the LCM subscription interrupt. When the robot receives messages on certain channels, it will wake up and update its path control state.

After updating states, the balanced bot is now able to follow commands from either the path published over LCM or the control signal given by the RC transmitter, to move as expected while maintaining balance. Control blocks are responsible for converting commands into desired forward and turning velocities of the robot and further into pulse width modulated(PWM) signals for both wheels.

B. State Machine and Decision Making Architecture

The state machine is used by the program to determine the next step that needs to be taken by the robot. The state of the robot is continously checked before setting any new forward or angular velocities. Users connect with the robot over $CONTROLLER\_PATH$ channel and send $robot\_path\_t$ message containing a sequence of poses (position and orientation) for the controller to achieve. Additionally, the robot can also be commanded to poses set in *botgui*.

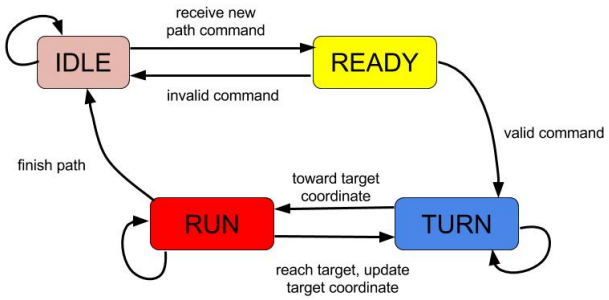

Fig. 5. Path Control State Machine.

Since the path received is essentially a sequence of corners at which the robot needs to translate to and then turn, a $Translate\ and\ Rotate(TR)$ finite state machine is required to ensure that the robot is performing the appropriate action and that it doesn't deviate from the set path. As shown in FIG 5, four states are defined and used for path control:

`IDLE` : The robot waits for path commands and the desired forward and turn velocity is set to 0. This is the default state when the robot is switched on and the robot accepts path commands only while in the `IDLE` state. Furthermore, while in the `IDLE` state the robot is commanded to hold a stationary setpoint. After all poses in the path command has been achieved, the robot state is set back to `IDLE`.

`READY` : The robot state is updated from `IDLE` to `READY` in the $robot\_path\_handler()$ when the robot receieves a valid path command. The path controller parameters are initialised and the current position is inserted at the top of the path command to generate the first waypoint. After successful initialization, the state would be updated to `TURN` in $bot\_motion\_control()$.

`TURN` : In this state, the robot would have a set desired forward velocity of 0 and it turns towards the next command location at a constant angular velocity till error in heading angle is less than 0.2rad. After successfully turning, it would wait untill forward velocity is actually 0 to stabilize, and then the state changes to `RUN`.

`RUN` : In this state, the robot follows the path and keeps moving till the robot is within 0.05m of the next pose location in the path command. Forward velocity and turning velocity in this state is calculated from the cascade control block described in section V.After reaching the target location, it will wait until the forward velocity becomes 0, and then the state updates to `RUN` and update the path index.

IV. Localisation and State Prediction

A. Odometry

The robot was primarily localised using odometry calculations to obtain its current position while driving. The incremental quadrature encoders affixed on each wheel gives us a phase shifted A and B signal with respect to ground and depending on whether A or B leads, we can determine the turning direction of the wheel. Consequently, the encoder counts either increase or decrease depending on direction of turning. We obtain the encoder counts over time step i, for left and right wheels as $N_{L,i}$ and $N_{R,i}$ every time the function $odometry\_get()$ is called by the controller during IMU interrupt. The gear ratio of the motors were determined by approximately rotating the wheel by one revolution and dividing the encoder count by the encoder resolution. The distance $d_{L,i}$ and $d_{R,i}$ travelled by left and right wheel respectively, is given by:

$$d_{L,i} = (\frac{\pi D}{nC})N_{L,i} \tag{1}$$

$$d_{R,i} = (\frac{\pi D}{nC})N_{R,i} \tag{2}$$

where D is the wheel diameter (0.08m), n is the gear ratio(34:1) & C is the encoder resolution (48 counts/revolution). Therefore, the displacement (m) and heading angle change (rad) of the centre of the robot can be calculated as:

$$\Delta d_i = (\frac{d_{R,i} + d_{L,i}}{2}) \tag{3}$$

$$\Delta \theta_i = (\frac{d_{R,i} - d_{L,i}}{2}) \tag{4}$$

The position of the robot in the world frame at time step i can be estimated as follows:

$$\begin{bmatrix} x_i \\ y_i \\ \theta_i \end{bmatrix} = \begin{bmatrix} x_{i-1} \\ y_{i-1} \\ \theta_{i-1} \end{bmatrix} + \begin{bmatrix} \Delta d_i cos(\theta_{i-1} + \Delta\theta_i) \\ \Delta d_i sin(\theta_{i-1} + \Delta\theta_i) \\ \Delta\theta_i \end{bmatrix} \tag{5}$$

The resultant state was published as an LCM message of type $pose\_xyt\_t$ on channel $ODOMETRY\_POSE$. The forward velocity v_i and angular velocity ω_i was also calculated, to be later used in controlling the robot.

$$v_i = \Delta d_i F \tag{6}$$

$$\omega_i = \Delta d_i F \tag{7}$$

where F (Hz) is the frequency at which the controller calls the $odometry\_get()$ function.

Additionally $theta\_regulate()$ function was implemented to ensure that the heading value remained between a circular range of $[-\pi, \pi]$. When $\theta_i > \pi$, $\theta_i = \theta_i - 2\pi$, when $\theta_i < -\pi$, $\theta_i = \theta_i + 2\pi$.

B. Gyrodometry

Odometry is based on the assumption that wheel revolutions can be translated into linear displacement but this assumption is invalid in certain cases. For instance, if one of the wheels in the differential drive robot slipped or encountered uneven surfaces, the encoders on the wheel would still register revolutions but the robot would not have linear displacement as modelled by odometry. To reduce this error, we also use gyroscope data from IMU to supplement the odometry. The yaw angle obtained from IMU was passed through a low pass filter to reduce spikes in measurements and then the difference

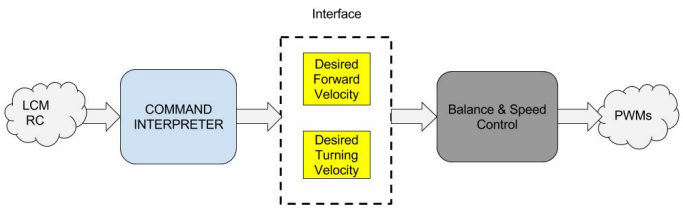

Fig. 6. Control Blocks Overview.

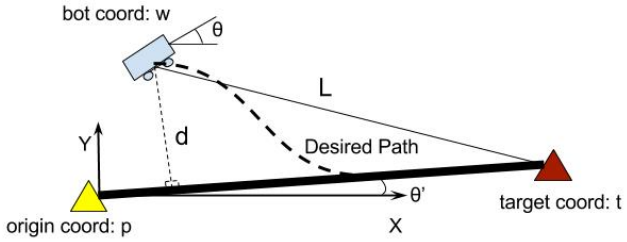

Fig. 7. Path Following Diagram

in yaw (heading angle) $\Delta\theta_{gyro,i}$ for time step i was calculated and compared to that obtained through odometry $\Delta\theta_{odo,i}$.

$$\Delta\theta_{gyro,i} = \frac{\omega_i}{F} \tag{8}$$

$$\Delta\theta_{GO,i} = \Delta\theta_{gyro,i} - \Delta\theta_{odo,i} \tag{9}$$

$$\theta_i = \begin{cases} \theta_{i-1} + \Delta\theta_{gyro,i} & if\ |\Delta\theta_{GO,i}| > \Delta\theta_{thresh} \\ \theta_{i-1} + \Delta\theta_{odo,i} & if\ |\Delta\theta_{GO,i}| \leq \Delta\theta_{thresh} \end{cases} \tag{10}$$

where $\Delta\theta_{thresh}$ is a user defined threshold that is determined by running a calibration sequence at the start of each trial while keeping the robot stationary and upright, and observing the bias in the gyroscope values.

The implementation of gyrodometry ensured that even when the robots position was disturbed while driving, the robot could recover its heading to a certain extent as it could identify that the wheel had slipped and the actual change in heading was different from that obtained through odometry. The gyroscope calibration sequence was required to be performed before every trial, as the gyroscope was very sensitive to both vibrations and temperature and the data from the gyroscope needed to be passed through a low pass filter before it could be used.

V. Control System

For control systems, a two-layer scheme is implemented as shown in Fig.6. The first layer of the control system is Command Interpreter, which is used to convert input commands into robot's desired moving velocity. The second layer is Balance and Movement Speed Controller, which enables the robot to move at a certain required speed while keeping balance. Output of the second layer is PWM for the wheels. The highest frequency for Command Interpreter block is 10HZ, and for Balance & Speed Control Block is 200HZ.

A. Command Interpreter

`LCM Command Interpreter:`

For path command received over LCM channel, we set desired forward velocity and desired turn velocity according to the state of the robot as determined by $bot\_motion\_control()$. In `RUN` state, we design the robot to move back onto the path even when it has been disturbed by implementing a path following controller in addition to the Cascade Control block as shown in Fig. 7 and Fig. 8.

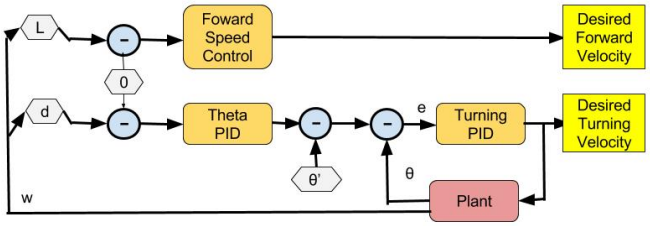

Fig. 8. Path Following Controller

In Fig. 7, θ denotes current heading direction of balanced robot and θ' denotes the path direction. w, p, t is robot's current position, start position and end position respectively in terms of (x, y). L is the euclidean distance between w and t. The robot's forward speed is set according to L so that it can start and stop smoothly. d is the distance from robot to the path, which is used to control the robot to move back to the path smoothly, as traced by the dashed line. d can be calculated as:

$$d = (t_x - p_x)(w_y - p_y) - (t_y - p_y)(w_x - p_x) \tag{11}$$

d is also used to mark the side upon which the robot is currently at with respect to the path. If $d < 0$ it denotes that robot is to the left of the path; if $d > 0$ then the robot is to the right of the path.

Fig. 8 shows the control block for path following. When there is an error in heading, the outer loop `Theta PID` calculates the desired heading direction of the robot according to path's direction and the inner loop `Turning PID` calculates desired turning speed for the robot to achieve the desired heading direction. Forward Speed Control blocks shown in Fig. 8 is a simple ramp function that ensures that the robot starts and stops smoothly. The maximum forward speed is limited to $0.2m/s$.

`RC Command Interpreter:`

For RC Command, it is much easier to convert the command values into velocities. RC commands consists of forward speed command and turn speed command, which are both in range of $[-1, 1]$. Fig. 9 shows the control blocks for RC command. Forward sensitivity and turn sensitivity is set as 0.2 and 1 respectively to limit robot's maximum speed. Turn Speed PID

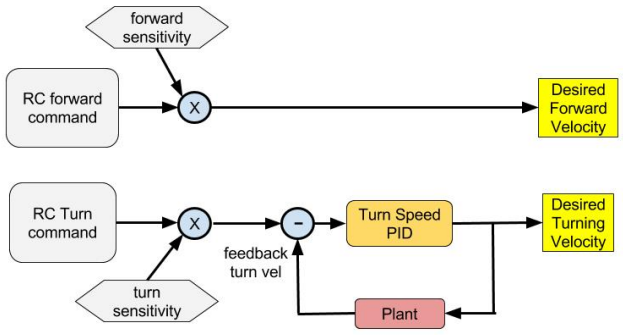

Fig. 9. RC Velocity Control.

is used to ensure that the angular velocity of the robot follows the RC command.

B. Balanced and Speed Controller

After interpreting the path commands into robot's desired movement velocities, a Balance & Speed Cascade Control block is used to convert the velocities into PWM signals for the wheels, as shown in Fig. 10. This control block is responsible for robot balancing and forward velocity control.

While Tilt Angle PID and Balance PID works together to find an angle at which the robot can balance and move forward at the same time at a desired forward velocity, the desired forward velocity is set by a ramp function that calculates the speed based on the distance between current and end waypoint. Balance PID operates at 200HZ to maintain stability and Tilt Angle PID operates at 40 HZ to enable the robot to stabilise at an angle, before giving a new command. If the frequency of the Tilt Angle PID were increased, when the robot tries to stop as it nears the end point, the forward velocity of the robot would decrease which would mean that the desired balance angle would hve to decrease to compensate for slower movement. This would in turn result in the robot having to accelerate to reduce angle which would be counterproductive

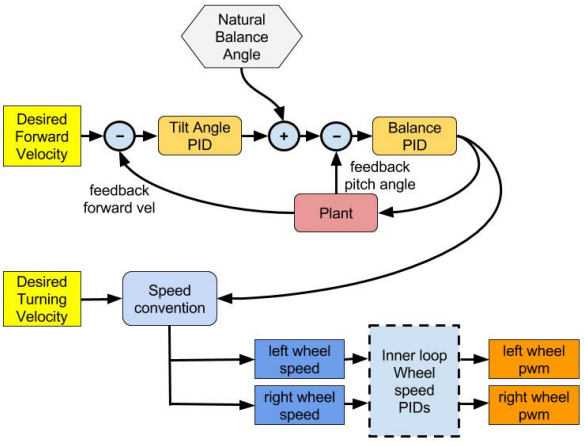

Fig. 10. Balance & Speed Control.

as then the error fed to the Tilt angle PID would increase. This ever increasing input and output would initiate the system in a positive feedback loop and the robot would never actually stop. The lower frequency at which the Tilt Angle PID is called ensures that the robot could stabilise its angle and thus reduce or increase its velocity before it is sampled again to find the error that feeds into the control block. The balance angle for steady state can be set by calibrating the IMU and looking at the bias in IMU when held upright but as this bias keeps varying every time we do a trial, the Tilt Angle PID actually compensates for this change by trying to maintain zero velocity and hence achieve the natural balancing angle instead of trying to achieve a set desired angle which may or may not be the natural balancing angle.

Desired forward speed V_f, desired turn speed V_t, speed for left wheel V_{wl} and right wheel V_{wr} can be calculated by a speed convention equation:

$$V_{wl} = V_f - V_t \cdot L_{wb}/2 \tag{12}$$

$$V_{wl} = V_f + V_t \cdot L_{wb}/2 \tag{13}$$

where L_{wb} is the length of the robot wheel base.

After obtaining velocities for each wheel, we convert them into PWM signals.A typical method used to control velocity through PWM is to implement inner loop PIDs. However, the problem we encountered while trying to implement inner loop PIDs is that generally 3 5 cycles are required for inner loops to become stable and meanwhile, the robot is not able to enter the balance loop. If we call the inner loop PIDs and Balance PID at the same frequency, the robot oscillates and never reaches a stable state.

Hence, we chose to adopt a simpler method to convert wheel velocities into PWM duty cycles. By recording the maximum wheel speed corresponding to 100% PWM duty cycle we calculated the ratio between duty cycle and wheel velocity. For both motors, the maximum wheel speed was recorded as approximately $1.6m/s$ when PWM is set to 1 and so we set the PWM duty cycle as $V_w/1.6$ and the PID degenerates to pure P controller.

C. PID Tuning Method

A proportional_integral_derivative controller (PID controller) is a control loop feedback mechanism (controller) that is commonly used in industrial control systems.A PID controller continuously calculates an error value $e(t)$ as the difference between a desired setpoint and a measured process variable, and applies a correction based on proportional, integral, and derivative gains. The output of the PID controller is the weighted sum of these three terms:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{14}$$

In our control system, up to seven PIDs are built and five of them are actually used. Most of the control blocks use only a PI or PD controller and their use would be discussed later in the section.

The ZieglerNichols tuning method is a heuristic method of tuning a PID controller. It is performed by first setting the I (integral) and D (derivative) gains to zero. K_p is then increased (from zero) until it reaches the ultimate gain K_u, at which the output of the control loop has stable and consistent oscillations. K_u and the oscillation period T_u are used to set the P, I, and D gains depending on the type of controllers used. The output of the PID values by ZieglerNichols method is calculated as:

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right) \tag{15}$$

T_i and T_d are typically set to $\frac{T_u}{20}$ to $\frac{T_u}{2}$ depending on the type of controller used. This method can approximately obtain K_p and a range of values for K_i and K_d. We further use a potentiometer to find the optimal value for each PID.

The various PID blocks that were previously mentioned for balance, speed and heading are listed below. Gains for the various PIDs are shown in TABLE 1.

`Turning PID`: This PID is used in the Path Following Cascade Control block for RC controlled driving. The output of this PID ensures that the robot is able to head in the same direction as the path even when disturbances are introduced manually (by holding a wheel, kicking the robot while moving) and run close to the desired path in a parallel fashion as shown in Fig. 11.

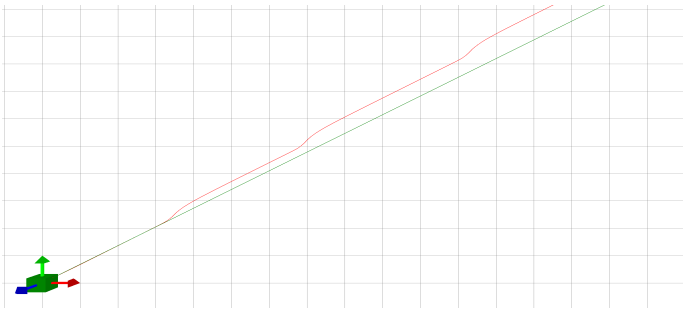

Fig. 11. Turning PID path.

`Theta PID`: This PID is used to control the heading direction of the robot to follow a desired path when the robot is driving autonomously in a square. Since we assume that the robot can respond quickly to any deviations due to the Turning PID and not overshoot to the other side of the path, a PD controller is used. When stable, d is 0 and $\theta = \theta'$ hence, I is unnecessary. After this PID, the robot can go back to the path when it is off track, as shown in Fig. 12.

TABLE I
CONTROL GAINS TABLE

Controllers	P	I	D
Turning PID	4	0	0.01
Theta PID	1.5	0	0.003
Turn Speed PID	0.04	0.001	0.0002
Balance PID	9.5	0	0.2
Tilt Angle PID	0.4464	0.0007	0.0099

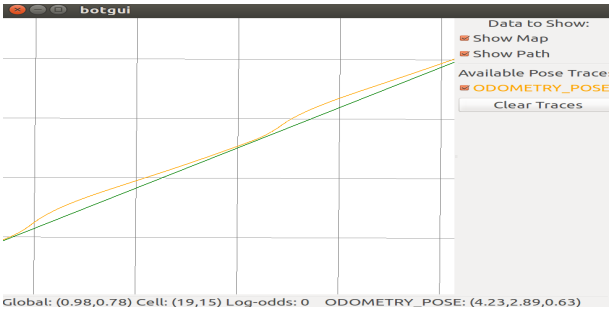

Fig. 12. Cascade PIDs path.

`Turn Speed PID`: This PID is used in RC control block to ensure that the robot follows turn speed commands. K_d is set to a very small value to prevent noisy RC commands. Note that both the Turn Speed PID & the Turning PID are used to control the robot turning velocity but the Turning PID sets the desired angular velocity and the Turn PID maintains set angular velocity and thus, they have different gains.

`Balance PID`: This PID is applied to keep robot's pitch angle at a certain setpoint. If the angle at which the robot naturally balances is found correctly, the robot is supposed to maintain balance and stay on spot. In stable state, the PID's output should be 0 when the robot's pitch angle reaches a set balance angle, thus, I is unnecessary in the PID controller.

`Tilt Angle PID`: As the natural balancing angle of the robot could channge due to bias and noise in IMU, hardware modification, collision and movements, it is not recommended to set it to a constant value. As a result, a PID controller is required to calculate the actual balance angle at any state. An integral value is required in the controller because when the robot is at a stable state, it reaches a desired forward velocity and the input to the PID becomes 0. The output tilt angle, however, is not equal to zero, so to help the robot maintain both balance and forward speed a bias iss required proportional to the velocity. D is set to a small value because while maintaining balance, the robot tends to oscillate quickly, which can cause a large D_term output that we wish to regulate.

VI. DATA ACQUISITION AND PERFORMANCE ANALYSIS

A. Optitrack Motion Capture System

Odometry and gyrodometry data depends entirely on encoder and IMU signals which are noisy and eventually the position calculated from them develops drift. To compare and

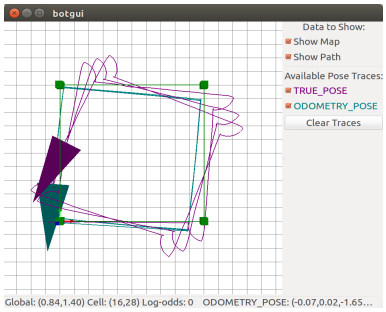

Fig. 13. Odometry with wheels mode trajectory map.

quantify the error accumulated in the system while driving, the robot was driven three times in a square in a room equipped with Optitrack and data from the Optitrack system was captured while the robot drove. The Optitrack system and cameras were calibrated using a wand with markers to cover the entire space detected by the cameras. After placing three markers asymmetrically on our robot to detect orientation and position, our robot was defined as a rigid body in Motive, the software used to calibrate and control the system. Data about our rigid body was then broadcasted over *robonet2* network and published as a LCM message over *TRUE POSE* channel. On the computer which was connected to the robot and *robonet2* network, we subscribe to both *TRUE POSE* and *ODOMETRY POSE* channel and display their traces on botgui.

B. Results

The following discussion details the results of the various tasks that were set for us and includes an analysis of the log data that was obtained from both the Optitrack and the robot.

1) Driving in a square with caster wheels: We command the robot (with caster wheels) to drive a 1m long square path three times to see the bias and error in odemetry and gyrodometry implementations. The robot's pose when estimated by odometry and the ground-truth pose from Optitrack can be seen from Fig. 13 and Fig. 14. The robot's pose when estimated by gyrodometry and the ground-truth pose from Optitrack can be seen from Fig. 15 and Fig. 16.

While driving in a square, it was seen that the robot stopped abruptly at each corner which jerked the caster wheel up momentarily before turning on spot. This behaviour was predictable at each corner and this also confirmed the fact that the robot consistently accumulated more drift and error in every consecutive round in a predictable manner. From the data, it can be seen that the errors for each round lie within a certain range and that the performance of the robot is better when gyrodometry is applied rather than odometry.

Wheel slippage over smooth surfaces causes error in orientation that results in increasing positional errors in odometry over time. As gyrodometry accounts for such slipping with its IMU data, the error in position is significantly reduced. However, we find that yaw angle still drifts over time. Even after calibration, the rate of drift in yaw can only be limited to a certain range and not completely eliminated.

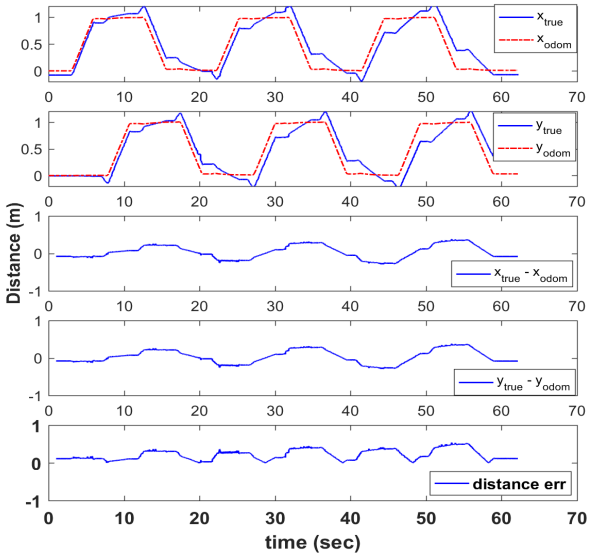

Fig. 14. Path and error plot of robot with odometry and caster wheel.

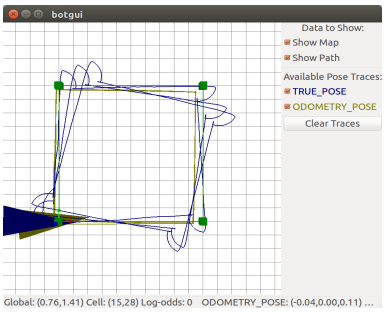

Fig. 15. Path trace of robot with gyrodometry and caster wheels .

2) RC Driving: Three test cases were used to test the effectiveness of the control scheme: holding a stationary setpoint, turning on spot and over a distance, and moving along a straight line. As we can see from Fig. 17, the pose estimated by odometry and the ground-truth pose error will increase when the robot is turning with a relatively large yaw angle change. This is also because of wheel slippage that was previously mentioned. It was found though that the robot motion control works better for forward and setpoint hold movement.

Setpoint Hold: In the beginning of the robot controller design, we tested the controller first in its ability to maintain balance at a fixed setpoint and then tested it during forward and pure turn driving segments. The setpoint hold function works well because of the cascaded nature of the controller which controls both the velocity that would be maintained at zero and the balance angle that would be indirectly affected by the velocity. The test results can be seen in Fig 18. There is a small constant error in the initialization. It is the odometry and optitrack error.

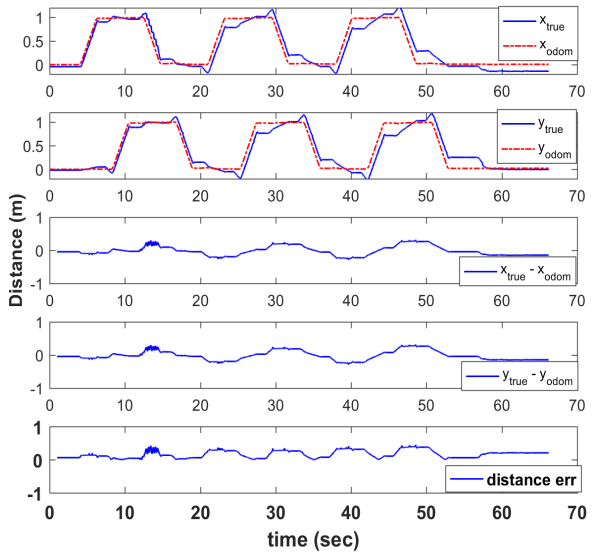

Fig. 16. Path and error plot of robot with gyrodometry and caster wheel.

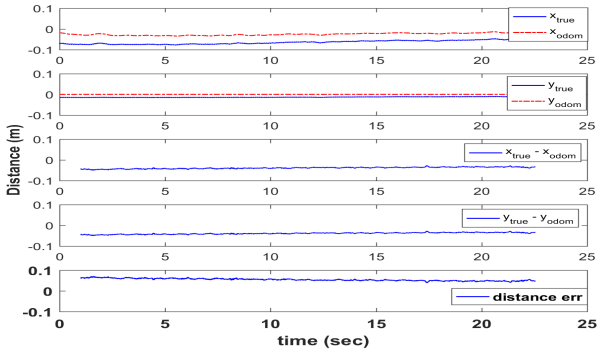

Fig. 18. RC controlled driving mode - Holding a etpoint

Drive Forward: As we can see from Fig. 19, the forward driving works well as errors due to heading change are avoided. This results in lower drift as the wheels do not slip as much when driving forward and not turning.

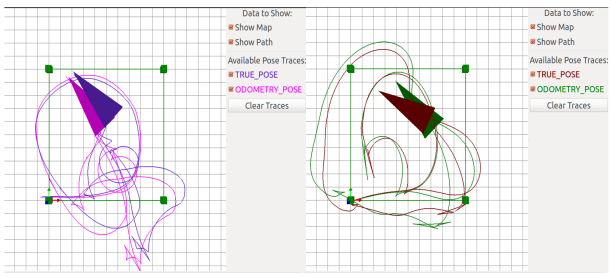

Fig. 17. Path traces for RC controlled driving mode - turning

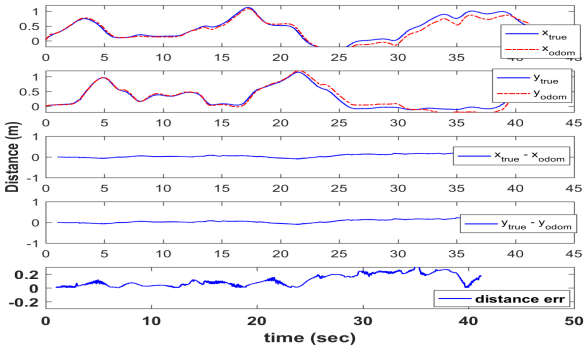

Fig. 20. RC controlled driving - turning

TABLE II
DRIVE SQUARE PERFORMANCE TABLE

	Maximum Err(meter)	**final theta err(deg)**
Odometry w/wheel	0.53	19.3
Gyrodometry w/wheel	0.32	5.2
Gyrodometry w/o wheel	0.26	29.5

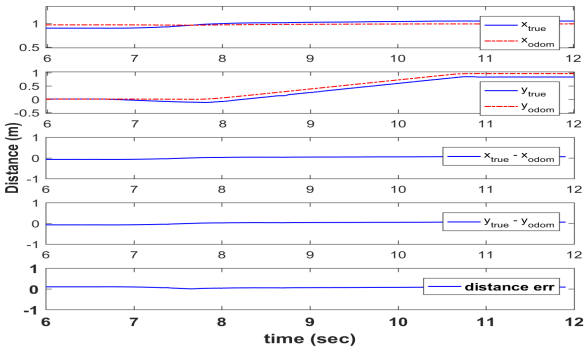

Fig. 19. RC controlled driving mode - pure translation.

Turn: As we can see from Fig. 20, the error in turning increases in part because of slipping and it was also observed that when the robot attempted to turn on spot it took a while for the wheel on the inner side to move at a constant speed that was opposite to that of the outer wheel. This slipping in the wheel causes the robot to slide back before turning as can be seen from the path traces in controlled RC driving. When compared to driving a square, RC control works better in part due to manual control and the intuitive compensation we perform.

3) Driving in a square without caster wheel: From Table. 2, we can find that the performance of driving in a square with a caster wheel and the performance of driving in a square without a caster wheel works better when gyrodometry is applied rather than pure odometry.

The performance of the robot when not equipped with a caster wheel is better and we attribute it to the reason that

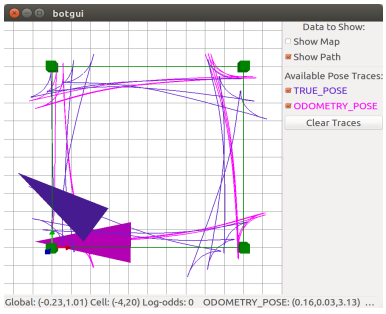

Fig. 21. Path traces for robot with balancing with gyrodometry

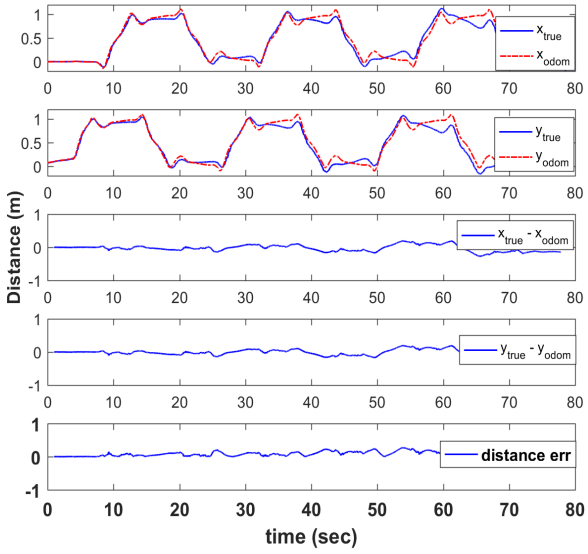

Fig. 22. Balancing and gyrodometry without caster wheel

the caster wheel in our design is not in the center, and the friction gives a bias to the turning velocity, which disturbs the controller. Since the accumulated bias for each square is constant and predictable, a reasonable turning speed offset and a small angle error check could enhance the performance of the square driving. However, the speed offset method doesn't make sense in real world indoor navigation. A better idea would be to use more reliable methods such as SLAM (Simultaneous localization and mapping).

C. Overall Performance

From the above discussions, we find that the robot can run correctly and stably according to the calculated Odometry or Gyrodometry pose data in either caster wheel mode or balanced mode. Even when driving off track, the robot can head back to the 'path' as shown by the traces in BotGUI. This indicates that the cascade path PID controller and Balance PID controller are working well. However, both Odometry and Gyrodometry methods cannot give the accurate position and heading of the robot. Even though results from Gyrodometry are much better than that of pure odometry, the accumulated error is still too much for real world indoor navigation. In conclusion, when we look into the cute balanced robot's inner world, we can reward it with a kitkat and say, nice job, you did it! But when we look at the real path that it traverses, there is still a lot of scope for improvement .

VII. Discussion

We optimized the hardware design to make the robot perform in a more stable and efficient way. We got better understanding of not only the software part, but the hardware design philosophy too. We also bring up a propeller design and discuss its advantages and limitations. Even though the control is not as accurate and well-tuned as we expected, the performance of the robot was robust when faced with disturbances. Future work would include replacing the ultrasonic sensor with other sensors like IMU (either independently connected to the Arduino board or using the data from Beaglebone through USB UART serial communication), a laser distance sensor, etc. and and evaluating their performance for a more robust system. As the ultrasonic sensor has obvious diffuse reflection, it sometimes overreacts to the environment noise. Overall, the performance of the Balancebot was satisfying in speed and stability.

Acknowledgment

Authors would like to thank Prof. Ella Atkins, course instructor, for scientific guidance, Dr. Peter Gaskell, lab instructor, for providing us with the proper equipment set and guidance throughout the experimentation process and lab sessions, Mr. Theodore Nowak, graduate student instructor (GSI) and Mr. Abhiram Krishnan, instructional aide (IA) for comments, insight and expertise that greatly improved our understanding of all hardware and software systems.

References

[1] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. Robot modeling and control. Vol. 3. New York: wiley, 2006.
[2] Sebastian Thrun, Wolfram Burgard, Dieter Fox. Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series)