MANCHESTER 1824

The University of Manchester

The Fault Tolerant Control of Autonomous System on Localisation

A dissertation submitted to The University of Manchester for the degree of MSc Advanced Control and Systems Engineering in the Faculty of Science and Engineering

Year 2018

Student ID: 9841876

School of Electrical and Electronic Engineering

Contents

Word Count: 8266

List of Figures

List of Tables

Abstract

The fault tolerant control is a popular solution for improving the availability of autonomous systems. It would be beneficial for robots if it can automatically detect and identify the fault and provide corresponding actions to minimize the expense. This project mainly implements the fault tolerant control in the non-holonomic robot Puzzle Bot which is made by the Autonomous Research Group in the University of Manchester.

The implementation consists of six procedures; implementation of the current sensor, the identification of the estimation motor model, the estimation by the measured current, the fault detection, the localisation task, and the localisation task with fault tolerant control. There is also a comparison of two types of estimations on the motor rotational speeds.

Acknowledgement

I would like to express my appreciation to my supervisor Dr Alexandru Stancu and the PhD Mario Martinez Guerrero for all their valued advice, constructive suggestions and supports during the last three months.

Declaration

I confirm that this project and dissertation are my original work, no content has been taken from work of others, and all referenced content have been cited and acknowledged.

Date: 3^{th} Sept 2018

Intellectual Property Statement

1. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the \Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
2. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.
3. The ownership of certain Copy right, patents, designs, trademarks and other intellectual property (the \Intellectual Property") and any reproductions of copyright works in the dissertation, for example graphs and tables (\Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions
4. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy, in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulation.

1. Introduction

Autonomous systems as mobile robots are currently in high demand in many fields such as automotive, aircraft, logistics, chemical processes and even daily use. They are often operated in unknown environment. Some area is restricted and cannot be entered; some is remote or extreme for people and needs to be explored. The design a robot from hardware to software requires skills on mechanical engineering, electrical engineering and computer science for essential tasks such as motion control, sensory feedback and signal processing.

In an unknown environment, localisation and mapping is an essential task for a robot. It first receives data of environments from embedded sensors. After proper signal processing, useful information is extracted to accomplish the localisation of the robot and mapping of the environment. In the case of given a goal, a position to achieve in a plane with axis of x and y, the motion control for the path planning then gives instructions to the actuators. Finally the robot performs any given tasks in such environment. These autonomous vehicles integrate electronic components which are usually expensive and fragile; in comparison with conventional vehicles, they can be difficult to repair.

One popular solution to prevent these situations is using fault tolerant control. It is often divided in two steps: fault detection and fault tolerant control. The basic theory of fault detection is to identify faults by comparing the real information from sensors with reliable estimations from a model. Fault detection can also be accomplished by implementation of external hardware. This method can deal with some hardware faults. Once the fault is detected, the robot can identify the cause of the fault and provide corresponding actions in time. If the replacement for failures is available, the executing task may not need to be terminated; else, the task must stop.

In this project fault tolerant control is applied to the tasks of Puzzle Bots on localisation and mapping. Puzzle Bots are non-holonomic robots made by the Autonomous Research Group in the University of Manchester. The pose of the robot is calculated by the speeds of the two drive wheels. Encoders and current sensors are implemented in the motor circuits.

Encoders are as transducer to calculate speeds; whilst current sensors calculate currents of motor inputs to estimate motor speeds. If the readings from encoders are distinct from the estimations speeds, faults will be reported and encoder measurements will be replaced by estimated speeds to finish localisation tasks.

For this project, *ROS* operating system will be used as the platform of communication among different procedures such as detecting speeds and currents, calculating estimations, motion controlling on motors, localisation, fault detecting and fault tolerant control. Those procedures work at the same time through the *ROS* environment and are communicated by messages.

1.1 Aims and Objectives

The aim of this project is to implement the current sensor, add fault detection model, develop the motion control and implement the localisation task with fault tolerant control on the non-holonomic robot Puzzle Bot. To achieve such aim following objectives are given

- Implement a current sensor in the single motor circuit and test the characteristics. Check the normal working condition of the current sensor in the circuit.
- Implement the mathematical motor model and tune it according to behaviours of the real motor. Figure out the property of every parameter in this model.
- Add currents from the current sensor to the motor model. Estimate motor speeds with measured currents.
- Make fault condition. If the encoder is broken, there will be a distinct difference between measured speeds from the encoder and the estimated speeds from currents.
- Add PID speed control for the motors so that the motor could rotate at a reference speed.
- Build the Puzzle Bot and implement current sensors.
- Build the localisation model. When receiving speeds, the model outputs the poses of the robot.
- Add the fault tolerant control in the localisation model. When a fault occurs, the measured speeds are replaced by estimated speeds to calculate poses of the robot.

2. Literature Review

2.1 Fault Tolerant Control

Fault Tolerant Control (FTC) is an emerging new area in the field of automatic control, for the need of developing the safety, reliability, availability, dependability of autonomous systems [1]. There are several methodological issues on FTC on the basis of the severity of faults' effects. The general procedure is to build a monitor system, diagnose the faults, analyse the effects and achieve fault-tolerant control [2].

A structural model of the robot system for estimation can present the relations among the system inputs and output variables as well as the parameters of the system. Robot plants in reality may have some nonlinear behaviour due to intrinsic constraints, which may increase the complexity of fault detection and tolerance [2].

Redundant hardware can be implemented to give alternative independent signals, which can be used as inputs of a mathematical structural model for estimations of outputs of the system. This type of method is called "analytical redundancy" [3]. The premise is that this redundant hardware gives reliable information and the mathematical model also matches the real system [3].

2.2 Kinematic Model for Non-holonomic Mobile Robot

Robots are usually designed to move on one specific environment. The working space of mobile robot is the land. The term holonomic means that the degrees of freedom are equal to the total degrees of freedom of the robot, which indicates that the robot could move in all directions at one time [5]. Non-holonomic robots will have some constraints on the mobility that the actuators cannot directly access [6] [7]. The robot that used in this project is the non-holonomic robot, which cannot move in the perpendicular direction of the side wheels. It only drives forward, backward, and makes turnings. This constraint leads to the robot having two degrees of freedom, which are the acceleration and the turning. The non-holonomic constraint is equation (2.2.1) [5].

$$\dot{x}sin\theta - \dot{y}cos\theta = 0 \qquad (2.2.1)$$

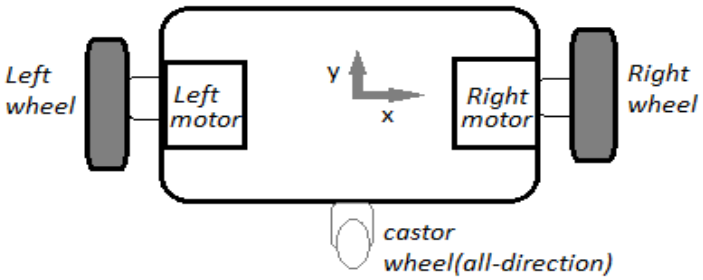


Figure 1 Schematic Presentation of the Non-holonomic Mobile Robot [5]

As Figure 2.2.1 shows, two side wheels are both drive wheels, and the castor is to keep the physical balance. The average velocities of the robot are calculated regarding the robot as ideal, referring to the centre of two drive wheels as the centre of the robot. And the transition relationship may be shown as from equation (2.2.2) to (2.2.4) [5].

$$\dot{x} = r\left(\frac{\omega_r+\omega_l}{2}\right)cos\theta \tag{2.2.2}$$

$$\dot{y} = r\left(\frac{\omega_r+\omega_l}{2}\right)sin\theta \tag{2.2.3}$$

$$\dot{\theta} = r\left(\frac{\omega_r-\omega_l}{l}\right) \tag{2.2.4}$$

r is the wheel radius, $\omega_r$andω_l are angular speeds of the right and left wheel respectively. θ refers to the turning angle of the robot and l refers to the distance between two side wheels. Considering the rigid feature, the robot velocity should be the average velocities of left and right wheels, and the turning angular speed should be the difference between the left and right wheel speeds [5]. When the kinematic model is used in codes of the project, the model is discretized as from equation (2.2.5) to (2.2.7)

$$\frac{\Delta x}{\Delta t} = r\left(\frac{\omega_r+\omega_l}{2}\right)cos\theta \tag{2.2.5}$$

$$\frac{\Delta y}{\Delta t} = r\left(\frac{\omega_r+\omega_l}{2}\right)sin\theta \tag{2.2.6}$$

$$\frac{\Delta \theta}{\Delta t} = r\left(\frac{\omega_r-\omega_l}{2}\right) \tag{2.2.7}$$

2.3 Encoder

Encoders are transducers that convert one type of information to another. Encoders for speeds sense the positions with reference to calculate the magnitude and orientation of the

rotational speed. The encoder is comprised of a LED, a disk with opaque and transparent segments, and a light detector. When the disk rotates with the shaft to be measured, the light passes those different segments, which could generate pulses to count. Two code tracks would make it possible to test the rotating direction of the shaft, in the condition of the segments of the two tracks are the same except that one of the track segments is 90 degrees lagging or leading [5].

2.4 Robot Odometry

Robot odometry is using rotational speeds to estimate the change with time on positions. This method is also called Dead-Reckoning Localization (or Motion-Based Localisation). According to the kinematic model in Section 2.3, the speeds could be calculated by using encoders. Therefore, the pose of the robot is integrated by adding the previous step with the discretized displacement, which is multiplied by step time and the velocity shown in Section 2.2 [6].

$$x_{k+1} = x_k + \Delta t \left(r \frac{\omega_r + \omega_l}{2} \right) cos\theta_k \tag{2.4.1}$$

$$y_{k+1} = y_k + \Delta t \left(r \frac{\omega_r + \omega_l}{2} \right) sin\theta_k \tag{2.4.2}$$

$$\theta_{k+1} = \theta_k + \Delta t \left(r \frac{\omega_r - \omega_l}{l} \right) \tag{2.4.3}$$

This method always brings with increasing random errors [5]. During the localisation, the calculation process on the pose is integration, and the odometry errors are also integrated with increasing distance; therefore, the growth of uncertainty of the pose is unbounded [6].

There are other methods for localisation, the common one is Map-Based Localization with exteroceptive sensors. There is a map with some known landmarks with information on the environment, and this type of sensors sense distance between the body and the landmarks to localise the position [5]. In this project, the sensors can only calculate speeds of robots regarding the reference frames to extract information for localisation.

2.5 DC Motor Characteristics

The basic principle of the DC motor is to transfer the electrical energy of the armature circuit to mechanical energy of the rotor by counter-electromotive force (back e.m.f.) as Figure 2.5.1.

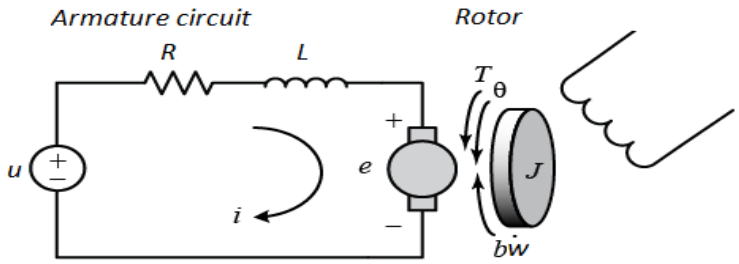


Figure 2 Schematic Presentation of the Motor Circuit [7]

According to Kirchhoff's law the equation of equivalent circuit is shown in Equation 2.5.1. V represents the armature circuit source voltage, e is the electromotive force (e.m.f.) that is transferred to mechanical system, R and L are the resistance and inductance of the armature circuit respectively [8].

$$V = e + IR + L\frac{dI}{dt} \tag{2.5.1}$$

The rotor is surrounded by permanent magnet with several poles. When the armature current passes through the coil wires, an electromagnet force can be generated because of the attraction or repulsion of the permanent magnet [8].

On the other hand, the motor torque is proportional to the armature current, and the counter-electromotive force is proportional to the angular speed of the shaft as equation (2.5.2) and (2.5.3) respectively. K_t represents the motor torque constant, and K_e is the electromotive force (e.m.f.) constant. In SI units, $K_e$and K_t are equal [8].

$$T = K_t I \tag{2.5.2}$$

$$E = K_e \omega \tag{2.5.3}$$

$$k = K_e = K_t \tag{2.5.4}$$

In steady-state conditions, the derivative value of I is zero, and there is a direct proportional relation shown as equation (2.5.5) between the torque T and the rotational speed ω according to Equations above (from (2.5.1) to (2.5.4)).

$$\omega = \frac{V}{k} - \frac{R}{k^2}T \tag{2.5.5}$$

This curve is shown in Figure 2.5.1 (a). The stall torque is the maximum while the motor is not rotating. After this torque, if the armature voltage still increases, the motor may rotate, with increasing rotational speed but smaller torque. The rotational speed is the maximum when no load is applied on the motor. The relationship between motor torque and speed is approximately linear as Figure 2.5.1(a).

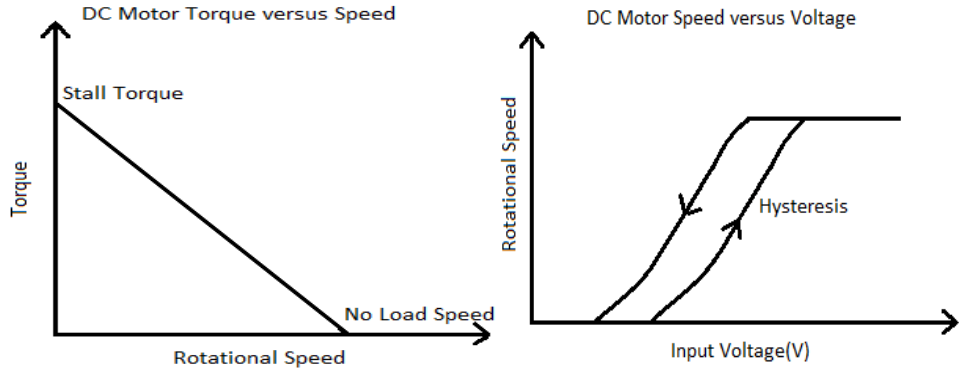


Figure 3(a) DC Motor Torque versus Speed (b) DC Motor Speed versus Voltage [7]

There also exists the hysteresis because of the magnets. The motor is magnetisation when the voltage increases from low to high, and from high to low is demagnetisation. These two situations might lead to different speed/voltage curves as Figure 2.5.1 (b) [8].

2.6 Mathematical Motor Model

The mathematical model of the motor is built in order to calculate the theoretical rotational speed from the armature voltage and current introduced in Section 2.5. The armature voltage u is V in Section 2.5, assumed as the input of the system, and the output is the rotational speed ω. The equation (2.5.1) can be transformed to equation (2.6.1).

$$L\frac{di}{dt} = -Ri - e + u \rightarrow \frac{di}{dt} = \frac{1}{L}\left(-Ri - K_e\frac{d\omega}{dt} + u\right) \tag{2.6.1}$$

It is also assumed that the rotor and shaft are rigid, and the damping ratio of the mechanical system is b. According to Newton's law, the second order dynamic equation of the rotor is as equation (2.6.2), where J represents the inertia of the rotor and m is the load torque [4].

$$J\frac{d\omega}{dt} + b\omega = T - m \rightarrow \frac{d\omega}{dt} = \frac{1}{J}(K_t i - b\omega - m) \tag{2.6.2}$$

Assumed that K_e and K_t are identical to k as equation (2.5.4). In this project, the discretised version is used as Equation (2.6.3) and (2.6.4) according to Equation (2.6.1) and (2.6.2).

$$i_k = \frac{dt}{L}(-R * i_{k-1} - K * \omega_{k-1} + u) + i_{k-1} \tag{2.6.3}$$

$$\omega_k = \frac{dt}{J}(K * i_{k-1} - b * \omega_{k-1} - m) + i_{k-1} \tag{2.6.4}$$

The armature voltage is applied by PWM signals from the motor driver.

2.7 *ROS* Operating System

ROS (Robot Operating System) is an open-source operating system for robots. It integrates the low-level device controls and software of the robot as nodes and services. In this way, the robot physical parts and missions are distributed into frameworks, and nodes are able to communicate by messages [9]. Every node could work independently as long as required information is known. Communicating from one node to another, messages could be different types and could be distinguished by topics. Messages of the same type are sent in one topic [9]. A whole task can be comprised of many nodes. The whole working state could be presented as a graph shown in Figure 2.7.1. Under *ROS* system, missions from different physical parts of the robot could be easily operated at the same time. And real time data could be managed as soon as possible. Every time the node receives messages from another node or topic, there would be an interrupt to execute a call-back function in the node.

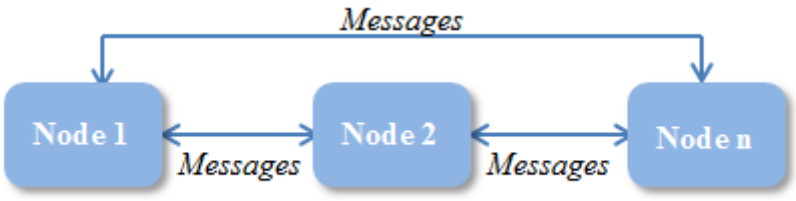


Figure 4 Graph Presentation of Nodes [9]

The supported languages of *ROS* are *Python* and *C++.* The basic approach of communication by *ROS* is using publishers and subscribers, where nodes could send messages out and receive messages in respectively. *ROS* has its own internal clock time in *ros::Time* class and could be used for sampling time. The Node spinning time is set by *ros::Rate* as *loop_rate()* [10].

2.8 Cumulative Moving Average Filter

The general equation of a cumulative moving average filter is equation (2.8.1). It is a good choice for time domain signals.

$$y = \frac{x_1+x_2+\cdots+x_N}{N} \tag{2.8.1}$$

y is the filtered output. From x_1 to x_N are all raw measurements. N is the filter size. Every measurement may have corresponding time point. If y is used to be the measurement of number N data, there might be a time shift for the signal [11].

3. Methodology

3.1 System Design

3.1.1 Hardware Implementation

The final project is implemented in the Puzzle Bot. A Puzzle Bot consists of two encoders, two drive wheels, a castor wheel, two motors, two switches, a puzzle plastic chassis, an *Arduino Mega 2560* board, a Raspberry Pi, a *TP-Link* router and a battery pack. Figure 5 is the robot schematic presentation with important wire connections. Current sensors and encoders are the two types of sensors for speed calculation. Current sensors are implemented serially in the motor input circuits. And encoders are implemented at the outputs of the motor. All sensor outputs are connected to Arduino board which is also on robot. Arduino is connected with Raspberry Pi, and Raspberry Pi communicates with computer through *ROS* messages.

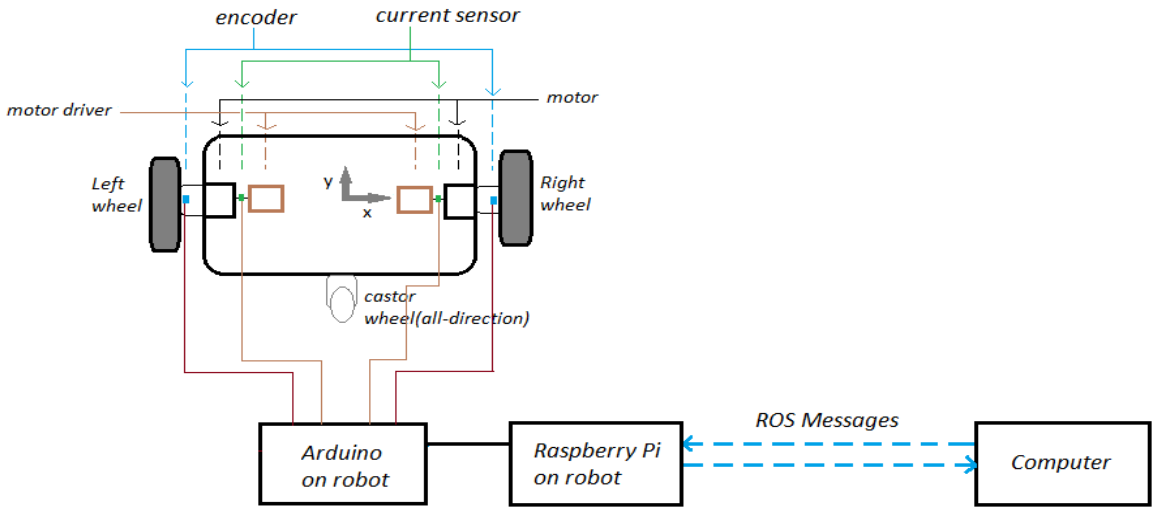


Figure 5 Schematic Presentation of the Hardware Implementation

Figure 6 Picture of Puzzle Bot

Two black wheels are used for driving and turning as Figure 6. Arduino and the router are charged by Raspberry Pi. Raspberry Pi is powered by the $12\ V$ battery pack. The radiuses of the wheels are both $0.046\ m$ and the distance between two drive wheels is $0.255\ m$ by measurement.

Arduino Mega 2560

The *Arduino Mega 2560* is a microcontroller board with 54 digital input/output pins, of which 14 are available as 14 outputs, 16 analogue inputs, $5\ V$ power pins, and 6 pins which are available for interrupts. These characteristics could completely meet the requirements when the board is connected with two motor encoders and current sensors: two $5\ V$ power pins, two analogue output pins, two ground pins for current sensors; four interrupt pins, two PWM output pins, two $5\ V$ power pins and two ground pins for encoders.

There are many examples and libraries available on Arduino IDE which could be executed for various functions. In the project, the servo library is used to send PWM signals to the motor driver by simply setting the duty ratio. Meanwhile, Arduino could be a *ROS* node when the *ROS* library is installed. The required data could be communicated with the computer and Raspberry Pi by publishing and subscribing topics in *ROS* environment.

Current Sensor *ACS712*

The Allegro™ *ACS712* is a full-integrated, precise current sensor which is based on low-offset Hall-Circuit [9]. The tested circuit is connected in serial with the current sensor as Figure 7.

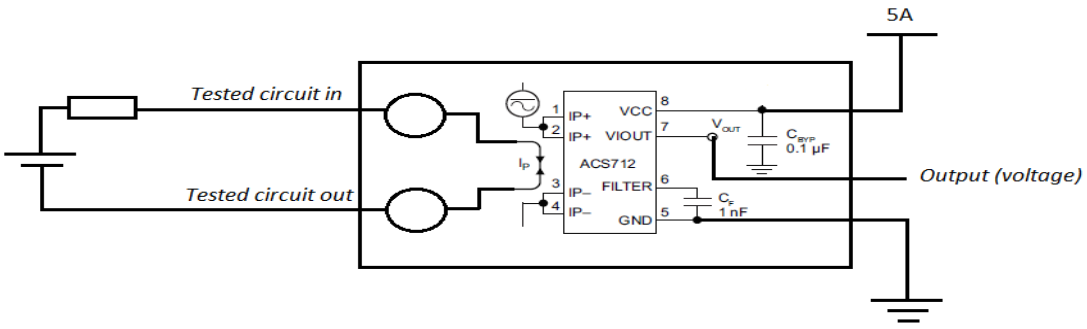


Figure 7 Schematic Presentation of the Current Sensor with a Tested Circuit

The current sensor will be in normal working condition if $pin\ 8$ is connected with $5\ V$ power and $pin\ 5$ is connected the ground. Under the operating principle of Hall-Effect, the tested circuit does not directly connect to the current sensor. The copper conduction of the tested circuit will generates a magnetic field, and the sensor circuit converts it into a proportional voltage. Normally, when the tested circuit side is off, the offset voltage at $pin\ 7$ is $2.5\ V$. The output voltage will proportionally change according to the magnitude of the tested circuit current.

The sensor type *ACS712ELCTR-05B-T* is used in this project with an output sensitivity of $185\ mV$ per amp changed on the tested circuit. The performance characteristics table in datasheet of the sensor shows that the noise is typically $21\ mV$ under the condition of $25℃$ environment temperature and $5\ V$ power voltage [9]. The principle of calculation the current of a test circuit is Equation (3.1.1).

$$current = \frac{V-2.5}{0.185} \tag{3.1.1}$$

In this project, this type of current sensor is connected serially with the motor circuit in order to test the current of the motor circuit in Figure 8.

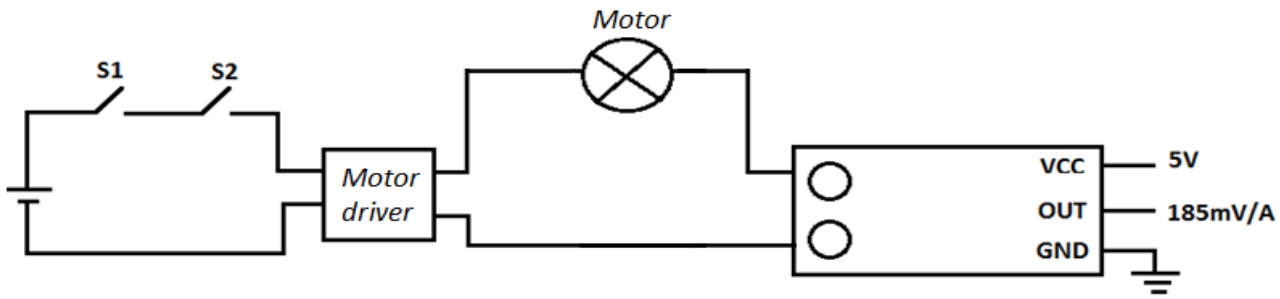


Figure 8 Schematic Presentation of the Current Sensor with the Motor Circuit

3.1.2 Software System Design

The whole system mainly uses C++ as the programming language (in Appendix 8.5). C# is used in the Arduino implementation file. The system is made up of five nodes, including Arduino, Model, PID, Fault and Localisation. The node Arduino is stored in the Arduino board; the node PID is stored in Raspberry Pi. The nodes *Model, Fault and Localisation* are stored in computer.

Each node has one main task in the program as $Table\ 1$ shows. All nodes work concurrently and communicate with each other by several types of messages through the topics in $Table\ 2$. They are started at the same time by using a launch file. Figure 9 shows the process of nodes passing messages.

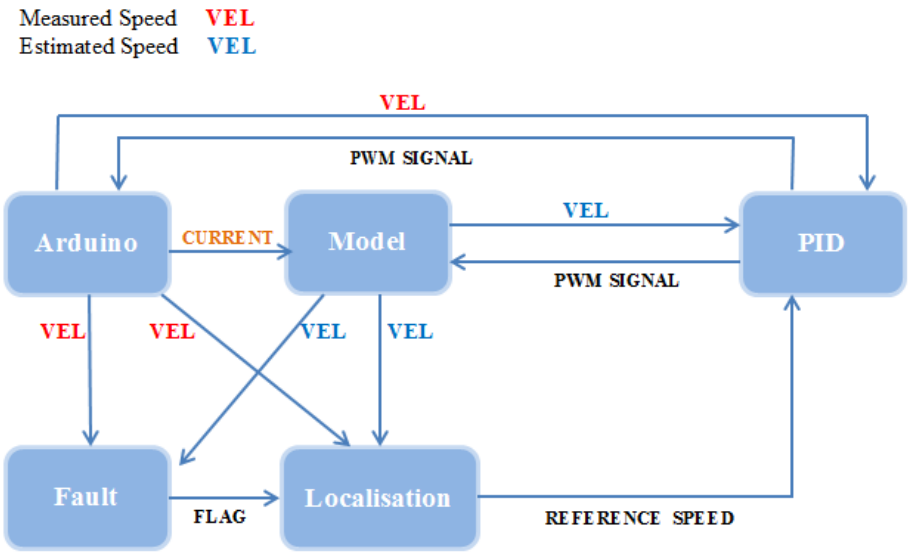


Figure 9 Diagram Presentation of ROS nodes

Table 1 Node Functions and Communications

	Function	Publish	Subscribe
$Arduino$	Read encoder speeds, read currents	Measured speeds, measured currents	Control PWM signal
$Model$	Estimate speeds from the mathematical motor model	Estimated speeds	Control PWM signal, measured currents
PID	Calculate the PWM duty cycle using PID control	Control PWM signal	Flags, reference speeds, measured speeds and estimated speeds
$Fault$	Fault detection	Flags	Measured speeds, estimated speeds
$Localisation$	Calculate the pose of the robot	The pose of the robot	Flags, measured speeds and estimated speeds

Table 2 Topic Functions and Communications

Topics	Message Inside	Position
$/Vel\_sens/wl$ and $/Vel\_sen/wr$	Measured speed from encoders	From the node $Arduino$ to PID, $Fault$ and $Localisation$
$/Vel\_ctl/wl$ and $/Vel\_ctl/wr$	Control PWM signal	From the node PID to $Arduino$ and $Model$
$/Vel\_ctl/SP\_V$	Reference Velocity	From the node $Pose$ to PID
$/Vel\_ctl/SP\_W$	Reference Turning Speed	From the node $Pose$ to PID
$/Vel\_cal/wl$ and $/Vel\_cal/wr$	Estimated speed from current and model	From the node $Model$ to $Fault$ and $Pose$
$/current$	Measured current from the current sensor	From the node $Arduino$ to $Model$
$/Vel\_warn/wr$	Flags	From the node $Fault$ to $Localisation$ and PID

3.2 Experiments

3.2.1 Motor Circuit Test with Current Sensor

The experiments of Section 3.2.1 to 3.2.4 all used the single motor circuit shown in Figure 10 and Figure 11. Similarly, the current sensor was connected serially at the motor input and powered by $5V$ signal from Arduino. Besides the power signals, there were three types of signals transmitted by Arduino. Encoder signals and current sensor signals were directly received by Arduino. PWM signals to the motor driver were sent by Arduino. The board was also connected with the computer with the USB cable.

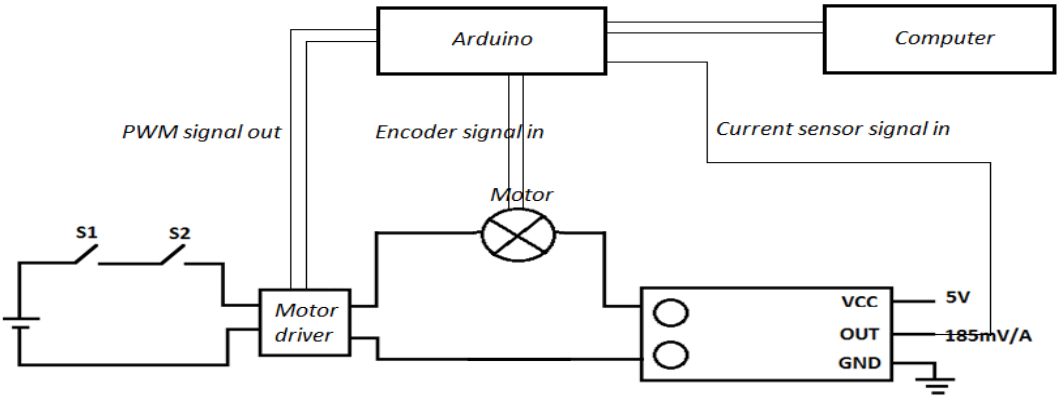


Figure 10 Schematic Presentation of Motor Test Circuit

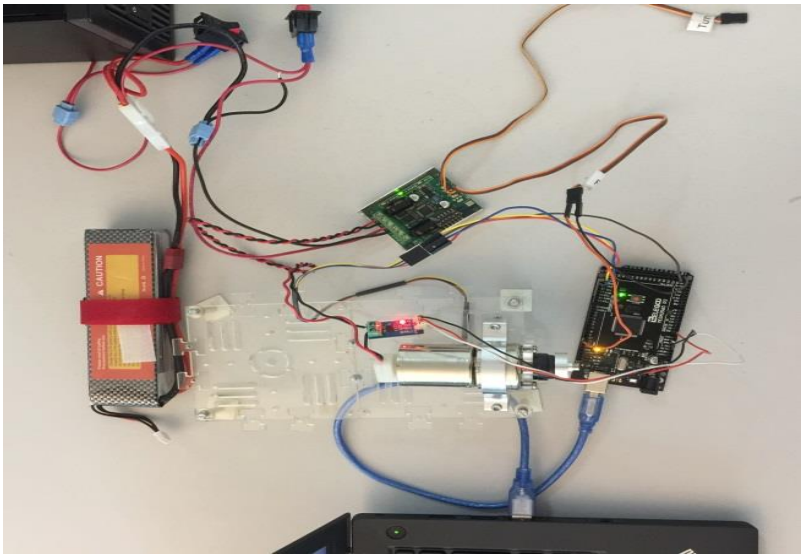


Figure 11 Picture of Motor Test Circuit

The physical motor circuit is shown in Figure 11. The Li-Po battery with a specification of $11.1\ V$ charges the motor driver. There are two switches between the battery and the motor driver. The motor is built on a plastic puzzle chassis fixed by four double-screw bolts

so that the wheel can rotate in the air. Four plastic pads are placed under the bolts to prevent the vibration of the chassis due to motor rotation.

In the computer A *ROS* package was built to receive and display the voltages and currents. The Arduino embedded file with *ROS* library is the node $/serial\_node$ in Figure 12 publishing the currents and voltages. Another node was built on the computer as a subscriber to receive the data. Figure 12 shows the working Node graph by using *ROS* function *rqt_graph*.

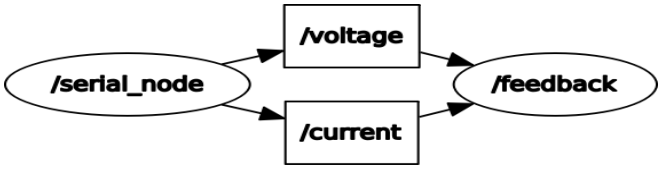


Figure 12 Node Graph of Sensor Tests

The Arduino board automatically discretised analogue signals by the default resolution 1024. The voltage of the sensor output was calculated by the Equation 3.2.1. The current of the motor circuit is calculated by Arduino using the Equation 3.1.1.

$$voltage = \frac{value}{1024} * 5 \tag{3.2.1}$$

One need for fault detection is estimation on speeds, which is relied on measured currents as inputs. It is important to reduce the noises of currents by filter so as to enhance the reliability of estimations. Meanwhile, when the motor starts changing states, impulses in currents may occur. These impulses are not supposed to be filtered.

The initial values of voltages without any input are about $2.5\ V$ and consequently the currents are within $0.5\ A$. Figure 13 (a) shows the test result when the node Arduino spinning once every $1\ ms$. After calculating, the noises of the currents are exaggerated because of scaling. Therefore, the noises of currents are bigger than those of voltages. The magnitude of noises of currents is about $0.1\ A$. The ripples of voltages and currents both increased when the motor was rotating.

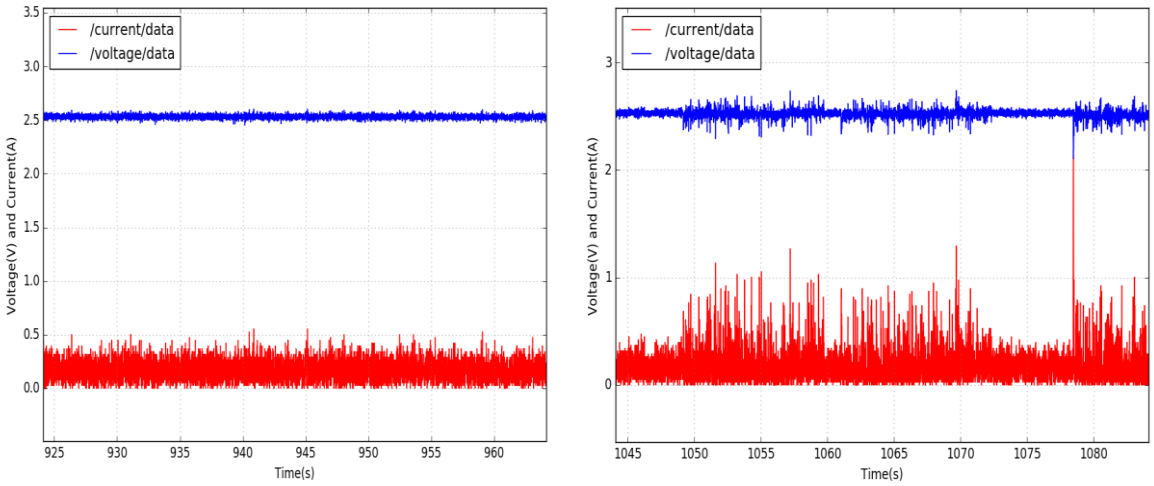


Figure 13 Raw Data from the Current Sensor (red – current; blue - voltage) (a) when the motor was still (b) when the motor was rotating

A cumulative moving average filter was added in the Arduino node as a low-pass filter for the current according to the Equation (2.3.4). The buffer size of the filter was 20, and the results shown in Figure 14. The time periods are consistent with those in Figure 13. There is a significant reduction in noises comparing Figure 14 with Figure 13. Figure 14 (a) shows that the average current is about 0.2A when the motor is still. When non-zero PWM signals are added to make the motor rotate, impulses clearly show as peaks in Figure 14 (b). These impulses appear when the motor motion state changes such as starting rotating or stopping rotating. The magnitudes of them were various and unsure. Besides the impulses, the average current is still at around $0.2A$.

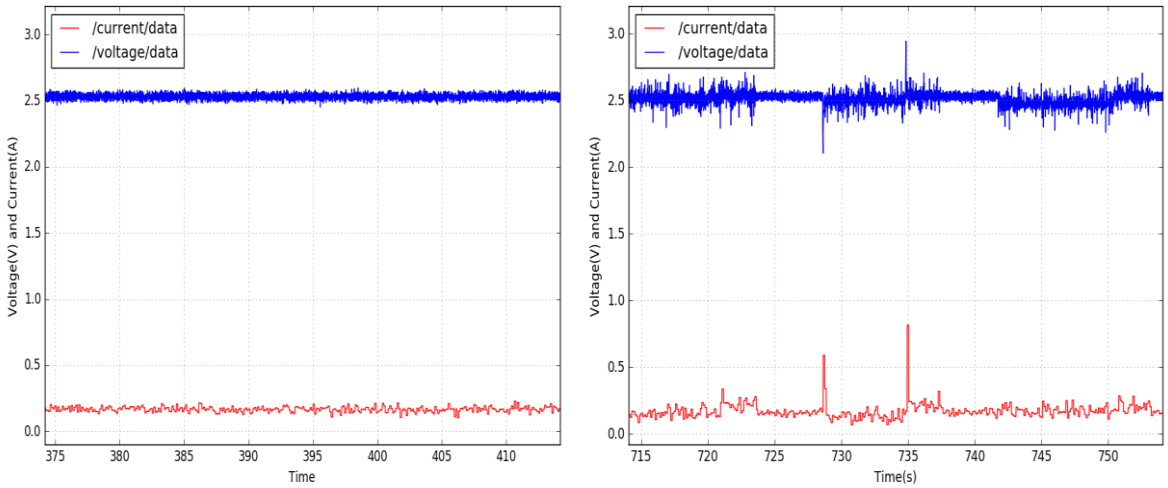


Figure 14 Current Sensor Data after Moving Average Filter (red – current; blue - voltage) (a) when the motor was still (b) when the motor was rotating

3.2.2 Motor Circuit Test with Mathematical Motor Model

In this section of experiments, the physical motor circuit is still the circuit in Figure 10 and Figure 11. In this section, the first test is the rotational speeds under different PWM duty cycle percentages. The node graph is in Figure 15. There is a temporary node /$rostopic\_24256\_1535487696124$ that publishes the duty cycle control signals through the topic $/Vel\_ctl/wl$. The node $/serial\_node$ in Arduino subscribes to this topic $/Vel\_ctl/wl$, receives the control signal messages, transfers them into PWM signals, and passes them to the motor driver. The motor driver then transfers PWM signals into corresponding input voltages to the motor circuit as the armature voltages.

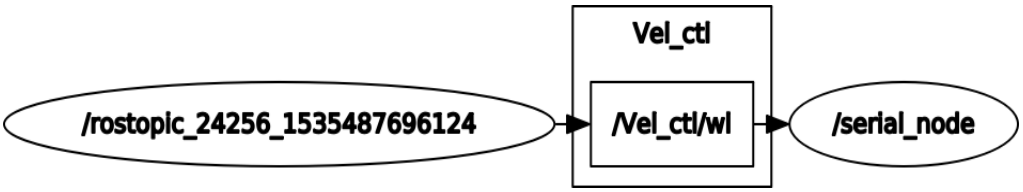


Figure 15 Node Graph of Motor Circuit Test

For reduction of the hysteresis, every single test has an interval of standstill of the motor without any input voltage. Figure 16 presents the curves of two independent tests on the relation between the RMS output voltages of the motor driver and the duty cycle ratios. The only slight difference between the two tests is on the battery voltage. The battery voltage of *Test 1* is $10.97\ V$, and the battery voltage of *Test 2* is $11.14\ V$.

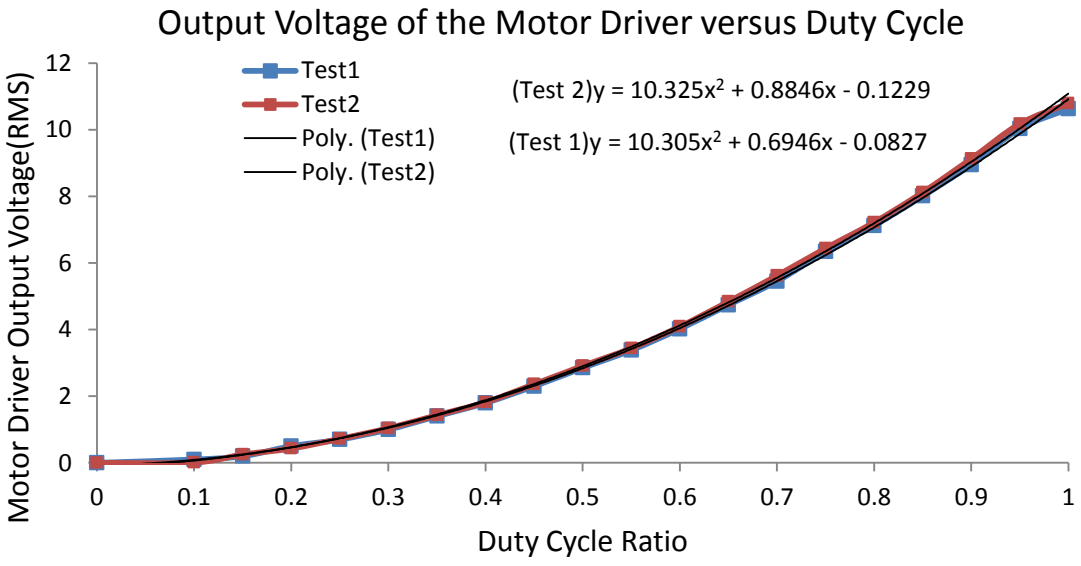


Figure 16 Output Voltage of the Motor Driver with Different Duty Cycle Percentages

The result curve is smooth and nonlinear, and can be approximated by a second order polynomial equation as Equation (3.2.3)

$$f(x) = 10.32x^2 + 0.8846x - 0.1229 \quad (3.2.3)$$

Meantime, under these duty cycle ratios, two independent tests on the motor rotational speeds are shown in Figure 17. The two curves are also nonlinear. There is a starting point for the rotational speeds. Due to the nature of hysteresis, the starting point was various within the range from 0.3 to 0.4 duty cycle ratio, but could not be measured precisely. After the starting point, the relation between the speeds and duty cycle is nearly linear.

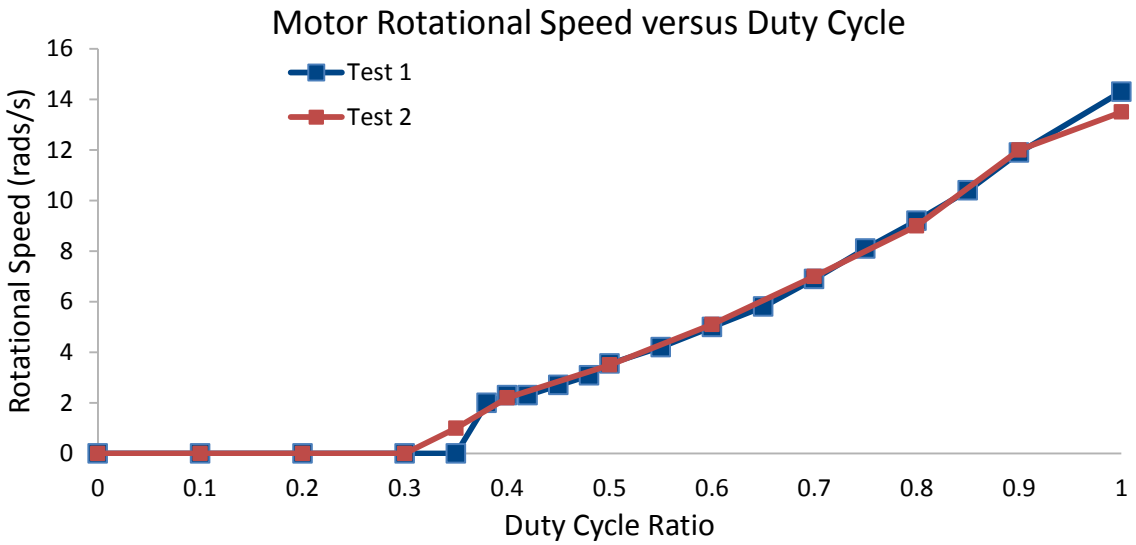


Figure 17 Rotational Speed of the Motor under Different Duty Cycle Ratio

The relations in Figure 16 and Figure 17 indicate that, from the duty cycle ratio, it is possible to estimate the motor rotational speed by the input duty cycle ratios using the mathematical motor model in Section 2.6.

The mathematical motor model was then built in the node $/feedback$ according to the model equations (2.6.3) and (2.6.4). The node graph is in Figure 18. There is also a temporary node $/rostopic\_21022\_1533742719849$ publishing the PWM duty cycle ratio signals through the topic $/Vel\_ctl/wl$. Both of the nodes $/serial\_node$ in Arduino and $/feedback$ in computer subscribe the topic to receive the duty cycle signal messages. The node $/feedback$ also subscribes to the topic $/Vel\_ctl/wl$ to receive the measured speeds from Arduino. The topic $/Vel\_cal/wl$ sends the estimated speeds.

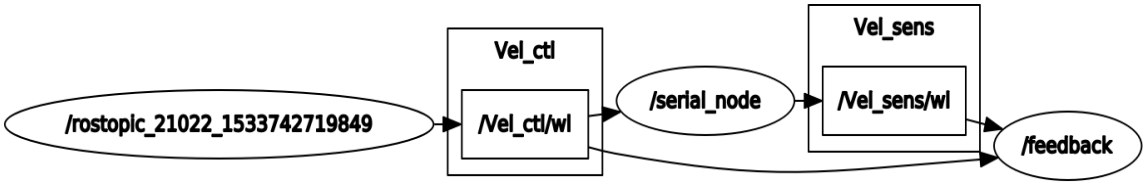


Figure 18 Node Graph of Motor Circuit Test with Mathematical Motor Model

The mathematical motor model was first built according to the recommended set of parameters from the Autonomous Research Group as shown below.

$$R = 8.7\Omega, K_e = K_t = \frac{0.03334Nm}{Amp}, J = 1.8e^{-6}kg \cdot m^2/s^2, L = 0.003H, b = 3.5e^{-6}Nms$$

The input voltage of the mathematical model was first set to be $11.1\ V$, the battery voltage, which implies that the duty cycle ratio should be 1. In this case, the motor is assumed to reach the maximum speed, which should be around $14\ rads/s$ according to Figure 17. The estimation results are shown in Figure 19. The estimated speed, whose steady state value is over $300\ rads/s$, is distinct from the measured speed from the encoder.

After receiving the message of 100% duty cycle, as Figure 19 shows, the measured current from the current sensor had a sharp impulse and reached the steady state of $0.2\ A$ in a time within $0.1\ s$. On the other hand, the initial value of the estimated current is $0\ A$. The steady state value is within $0.05\ A$. Both of the speed and current responses are faster than the measured ones but with a delay in $0.1\ s$.

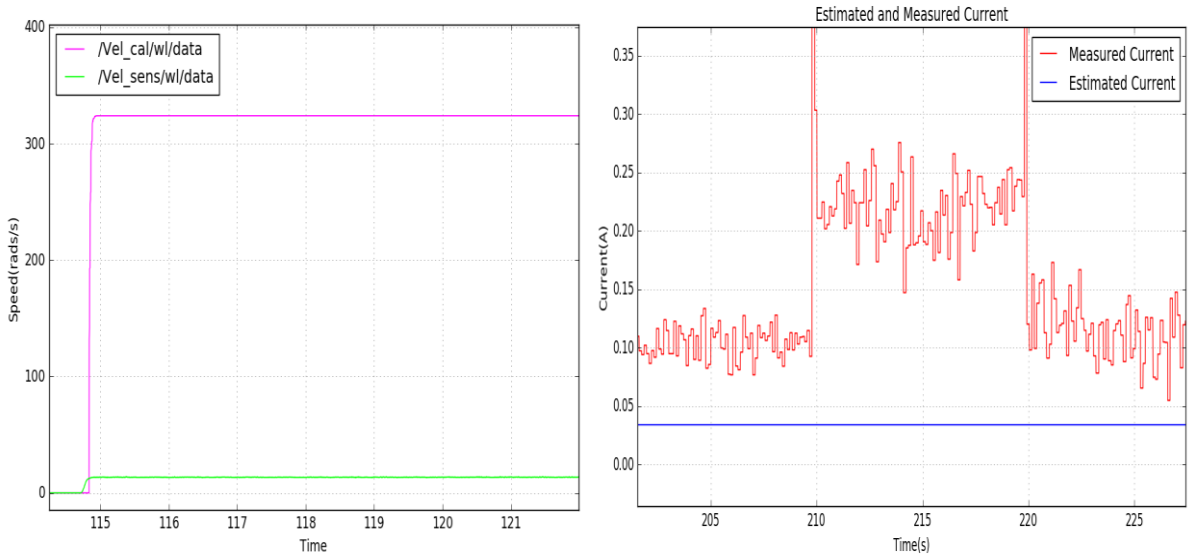


Figure 19 Estimation Results with Recommended Set of Parameters
(a) Voltages Comparison (b) Currents Comparison

Since the estimation completely mismatches the measured data on speed and current, the parameters should be tuned properly to improve the reliability of this mathematical motor model. Using Single Variable Method, the effect from every parameter may be discovered. $Table\ 3$ lists the cases of sets of parameters with only changing one parameter compared with $case\ 1$.

Table 3 Sets of Parameters of the Mathematical Motor Model

	$R(\Omega)$	$K(Nm/Amp)$	$J(kg \cdot m^2/s^2)$	$L(H)$	$b(Nms)$
$Case1$	8.7	0.03334	$1.8e^{-6}$	0.003	$3.5e^{-6}$
$Case2(J)$	8.7	0.03334	$1.8e^{-4}$	0.003	$3.5e^{-6}$
$Case3(b)$	8.7	0.03334	$1.8e^{-6}$	0.003	$3.5e^{-4}$
$Case4(L)$	8.7	0.03334	$1.8e^{-6}$	0.3	$3.5e^{-6}$
$Case5(K)$	8.7	0.3334	$1.8e^{-6}$	0.003	$3.5e^{-6}$
$Case6(R)$	1.7	0.03334	$1.8e^{-6}$	0.003	$3.5e^{-6}$

The criteria of response performance evaluation are checking their settling times, steady state errors, and overshoots. The detailed results are shown in Appendix 9.2. During the test, there was a trend that the overshoots grew with the addition of induction L, electromotive constant K, and armature resistance R. Meantime, the settling time was increased by the inertia J as well as the induction L. Only the addition of the damping

constant b and electromotive K would reduce the steady-state value of the estimated speed.

The objective is to tune the estimated speed to about $14.3\ rads/s$ without overshoot or much setting time. The damping constant b is then increased and k is reduced correspondingly. The tuned set of parameters is

$$R = 38\Omega, K_e = K_t = \frac{0.01Nm}{Amp}, J = 1.8e^{-6}kg \cdot m^2/s^2, L = 0.003H, b = 1.1e^{-4}Nms$$

With the tuned set of parameters, the result of speed response is shown in Figure 20. The steady-state value is almost $14\ rads/s$ with minimal settling time and no overshoot. The delay time between the estimation and measurement is ignorable.

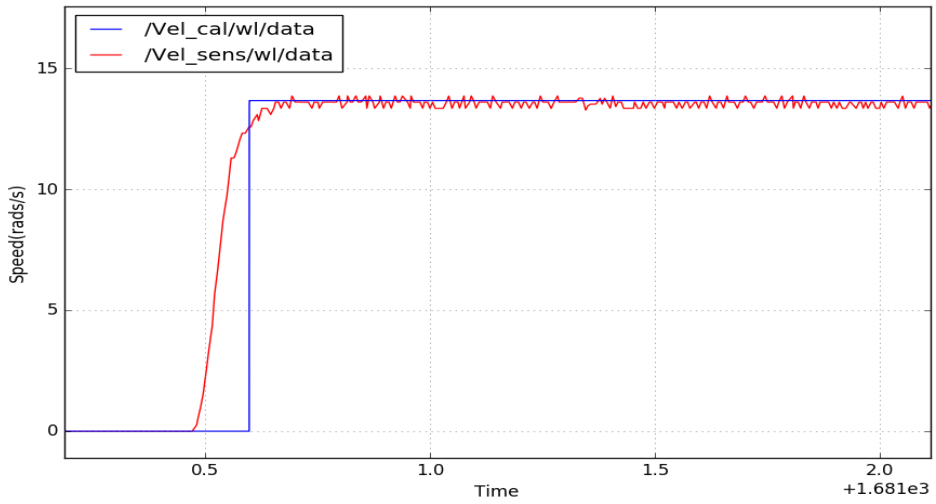


Figure 20 Measured Speed and Estimated Speed when Duty Cycle Ratio is 1

The result of comparison between estimated speeds and measured speeds from the encoder is shown in Figure 21. In Equation (3.2.3), the relation between the duty cycle ratio and the rotational speed could be tested. The result of comparison between estimated speeds from the model and measured speed from the encoder is shown in Figure 21.

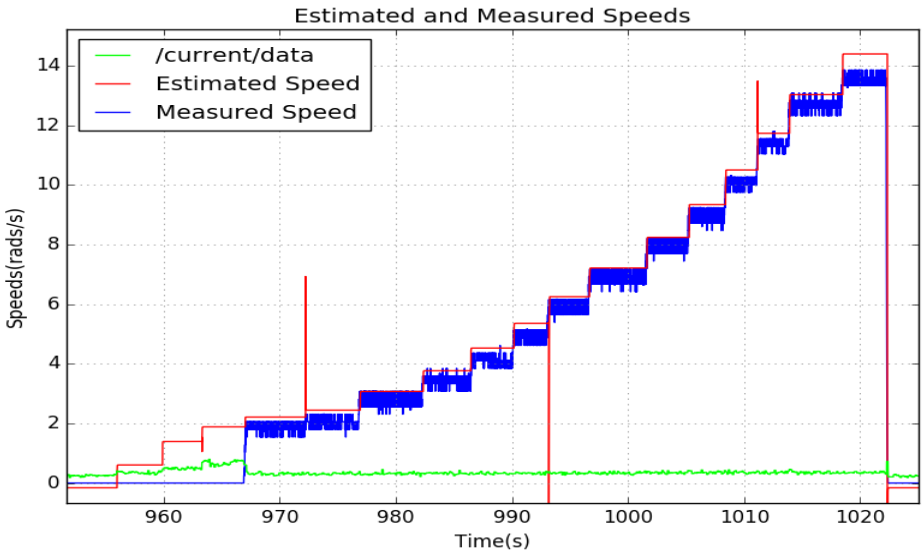


Figure 21 Measured Speed and Estimated Speed under Different Duty Cycle

This estimation in Figure 21 with the tuned set of parameters of the motor model almost matches the measured data. There is still a starting point for the motor resulting in a difference between the measured speeds and estimated speeds, when the input duty cycle is so small that the motor does not rotate. One solution is to set a segmented function for the estimation. The input could be changed to a fixed speed rather than duty cycle ratios. In this case, a PID speed control was added (in Appendix 8.3).

3.2.3 Motor Circuit Test with Mathematical Motor Model and Current Sensor

Considering the accuracy of data from the multimeter, using the estimated armature voltages limits the reliability of the estimated speeds. In this section of experiments, measured currents from the current sensor are directly used to estimate the motor rotational speeds. Therefore, only Equation (2.6.4) on speed estimation is used.

The parameters applied were the same with the tuned set of parameters in Section 3.2.2.

Figure 22 shows the correlation between the measured current from the current sensor and estimated currents using the model Equation (2.6.3) under different duty cycle ratios.

The duty cycle ratio was continuously increased from 0.1 to 1 and turned off to 0. The current is $0.25\ A$ when the duty cycle is 0. The motor did not rotate at first two duty cycle ratios which were 0.2 and 0.3. The current under the ratio of 0.3 is higher than $0.5\ A$. At other times, including when the duty cycle is 0.2, and when the motor is rotating, the measured currents are nearly stable at around $0.35\ A$.

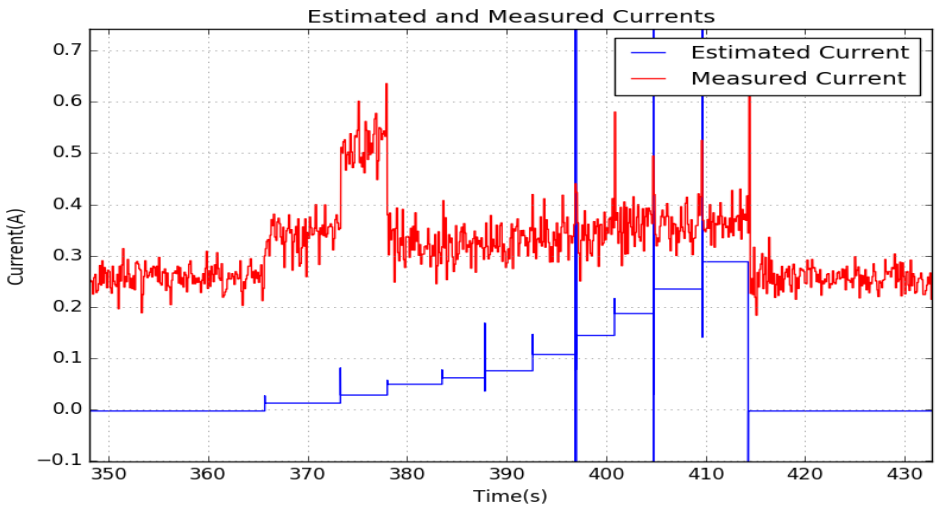


Figure 22 Estimated Current from the model and Measured Current from the Current Sensor (red – measured; blue – estimated)

The objective is to tune the model so that the steady-state error between the measured current and the estimated current is close to zero when the motor is still, whilst the noise is within a proper limit. Theoretically, the current should be zero when the armature voltage input is zero. The steady-state error of the measured current is from the offset error of the armature voltage. One method that used is to subtract the steady state error directly before integration.

On the other hand, due to the integration in the Equation (2.6.4) , the magnitude of the noises of the estimated speeds is much larger than those of measured currents. The electromagnet constant k and the damping constant b could be slightly tuned to reduce the magnitude of the noises. This set of parameters is as below.

$$R = 38\Omega,\ K_e = K_t = \frac{0.0095Nm}{Amp}, J = 1.8e^{-6}kg \cdot m^2/s^2\ , L = 0.003H, b = 2.0e^{-4}Nms$$

As Figure 23 shows, the blue lines represent the speeds estimated by the measured current, and the red ones are measured speeds from the encoder. The magnitude of the noises on estimation is within $8\ rads/s$.

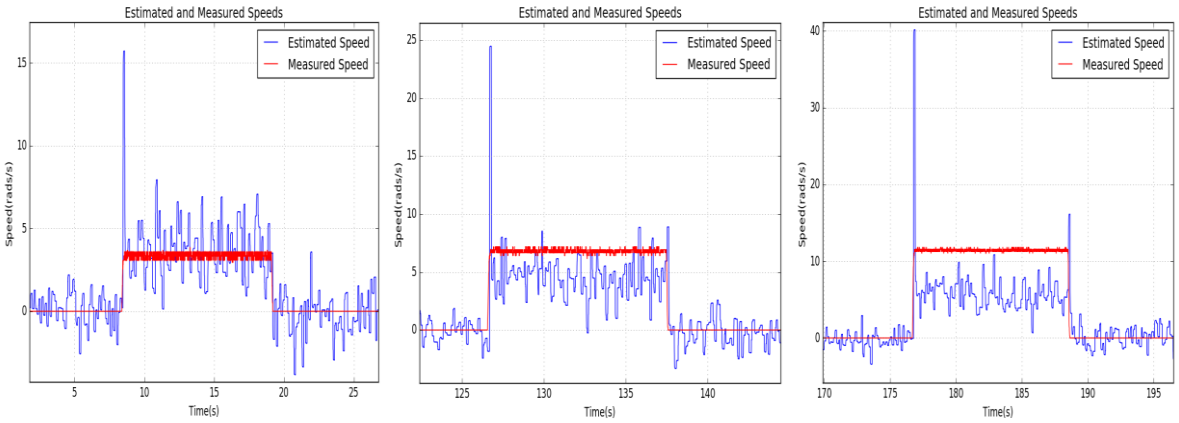


Figure 23 Estimated and Measured Speeds (red – measured; blue - estimated) (a) when the duty cycle is 50% (b) when the duty cycle is 70% (c) when the duty cycle is 90%

There is a potential problem that the estimated currents may not change with changing duty cycle ratio when the motor is always rotating. According to Figure 24, when the motor is turned on from static, the estimated speed rises from zero, while the average speed does not with the duty cycle ratios.

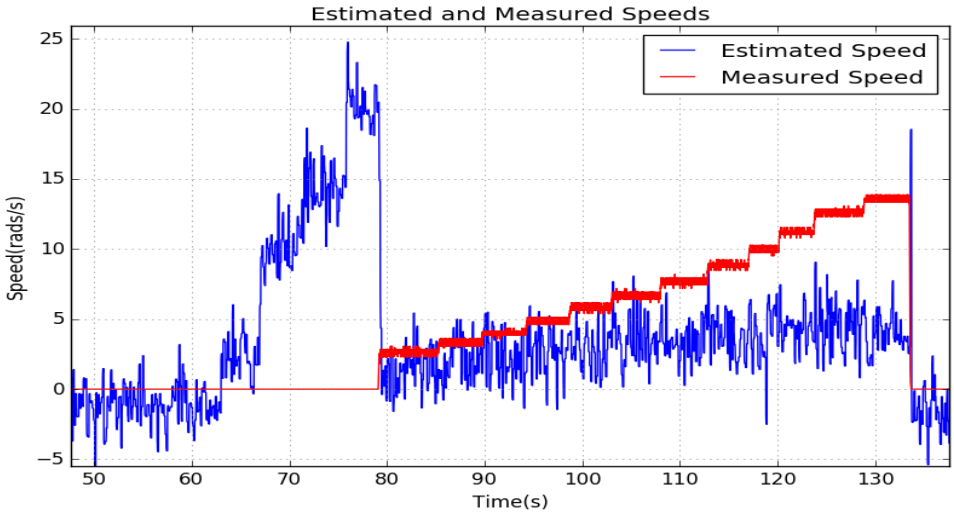


Figure 24 the Speed Estimated by Currents from the Current Sensor and Measured Speed from the Encoder (red – measured; blue - estimated)

3.2.4 Motor Circuit Fault Detection Test

According to the results of Section 3.2.3, the measured currents from the current sensor could test whether the encoder is in working condition. Figure 22 shows that the current may not change much when the motor is accelerating, while this current is undoubtedly more than the current when the wheel is still without any voltage input.

The estimated speed, which is calculated by the measured current from the current sensor, is therefore changed with the current. Figure 25 shows the estimated and measured speeds when the wires on the encoder are pulled out and connected in to imitate the faults on the encoder. The measured speeds become zero while the estimation is still above zero.

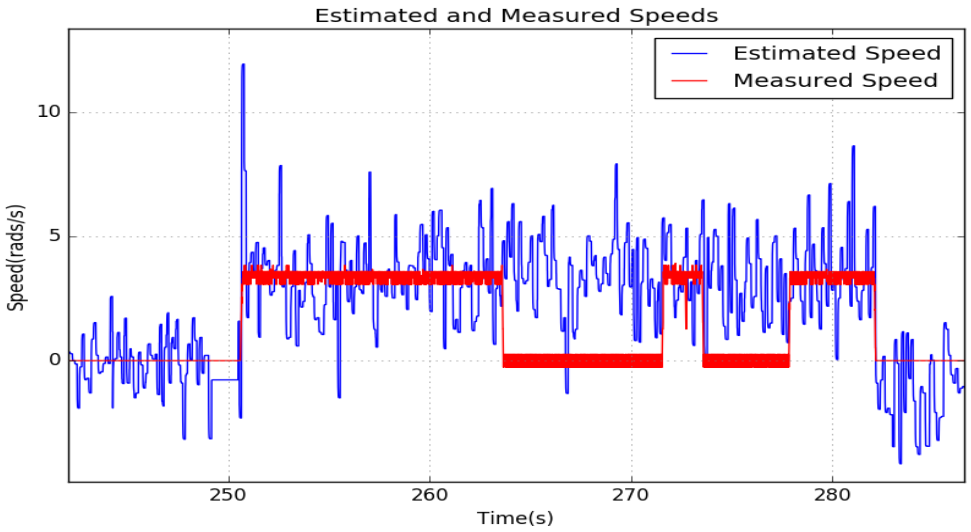


Figure 25 Estimated Speed and Measured Speed when the Encoder is Pulled out

The node *Fault* is for checking these faults above. Figure 26 shows the result after the implementation of fault detection. The fault case is that when the measured speed is zero, while the estimated speed by currents stays at the average value of $10\ rads/s$. In this case, the flag is on.

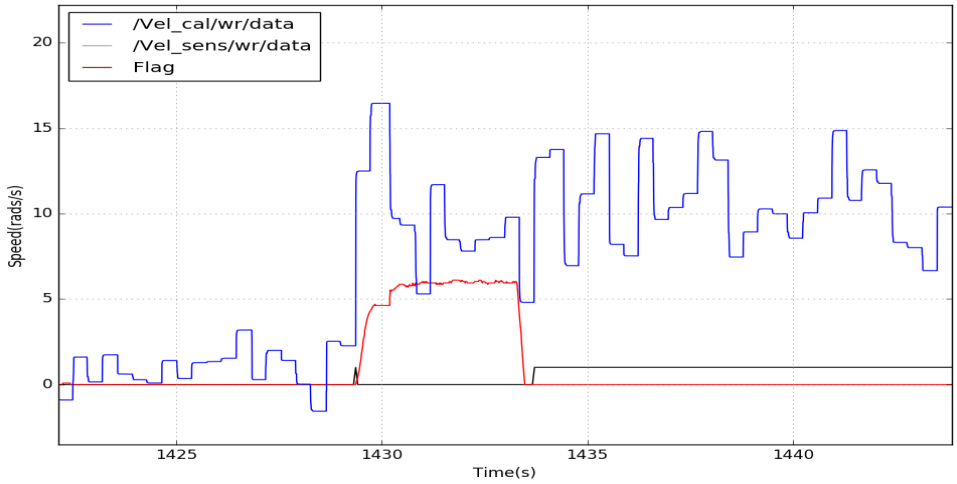


Figure 26 A Flag is Added when the Encoder is Pulled out

3.2.5 Puzzle Bot Localisation Test

The localisation test in this section is based on the dead-reckoning localisation method introduced in Section 2.2 and 2.4. The basic task on localisation is the pose estimation when the robot is driving in a straight line. The inputs are the set-point velocities and rotational speeds of the robot. They are also sent to the node PID to calculate the PWM duty cycle signals. For achieving a trajectory of a straight line, only the velocity was non-zero, with zero rotational speed of the robot. The result is shown in Figure 27.

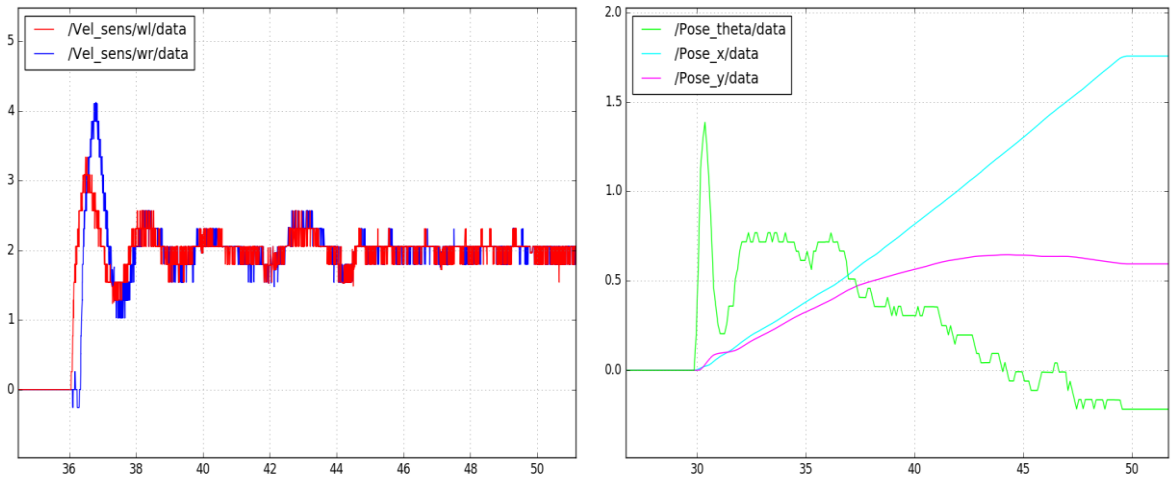


Figure 27 Speeds and Pose Results of Localisation Task of Straight Line (a) Measured Speeds of wheels (red – left; blue - right) (b) Pose of the Robot (blue – x; red – y; green - theta)

With the same PID parameters, the speeds of both wheels have a slight difference, which causes a small value on the turning angle of the robot, which indicates that there is a small drift on the robot instead of driving straight.

3.2.6 Puzzle Bot Localisation Test with Fault Tolerant Control

This section is the final step of this project. The fault tolerant control, the built estimation model, the fault detection, and motion control were all added on the localisation test shown in Section 3.2.5. The node graph is as Figure 9 shows. The expected outcome is that the robot keeps straight during the whole test regardless of the fault.

The first step is to implement the current sensor on the motor circuit of the right wheel of the robot. The mathematical motor model is implemented for both wheels. When driving straight, ignoring the drift error, the PWM duty cycle signal should be the same. Therefore, a comparison between the two types of estimations is possible at the same time. The left

motor is built in the full mathematical motor model, whose input is the estimated armature voltage; and the right motor is implemented in the model with the current sensor, whose input of the model is the measured current. The result is shown in Figure 28, where a duty cycle ratio 0.7 is input. The estimation using the full mathematical model with the estimated voltage is closer to the measured speed compared with the estimation using the measured currents.

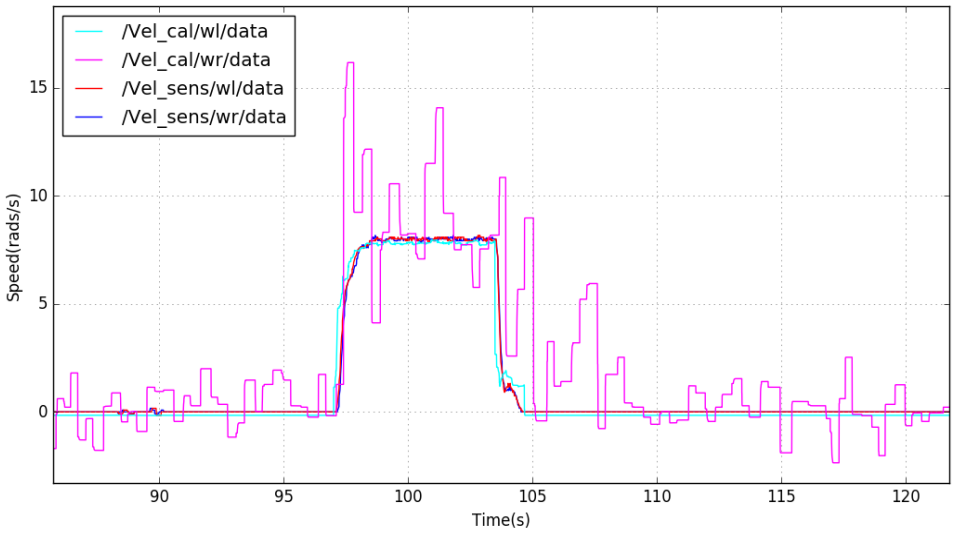


Figure 28 Estimated and Measured Speeds of Left and Right Wheels

Under the premise that the estimation is working regularly, the last step is to implement the fault detection and the fault tolerant control into the localisation task. When the encoder is broken in the process of the robot driving straight, the robot can detect the fault and replace the measured speed by estimated speed to continue the localisation task. The fault was generated by pulling out the connection wires of the right wheel encoder. The results are shown in Section 4.

4. Results

In Figure 29, the speed is estimated by the measured current on the right wheel. The result in Figure 29 shows that at the time that the encoder wire is pulled out, the measured speed suddenly goes to zero, the flag is on, and calculation on the pose is then changed. The pose is calculated by the estimated speeds. A distinct difference could be found that the pose calculated by measured speeds from encoders is smooth, while the calculated pose from estimation data is curved.

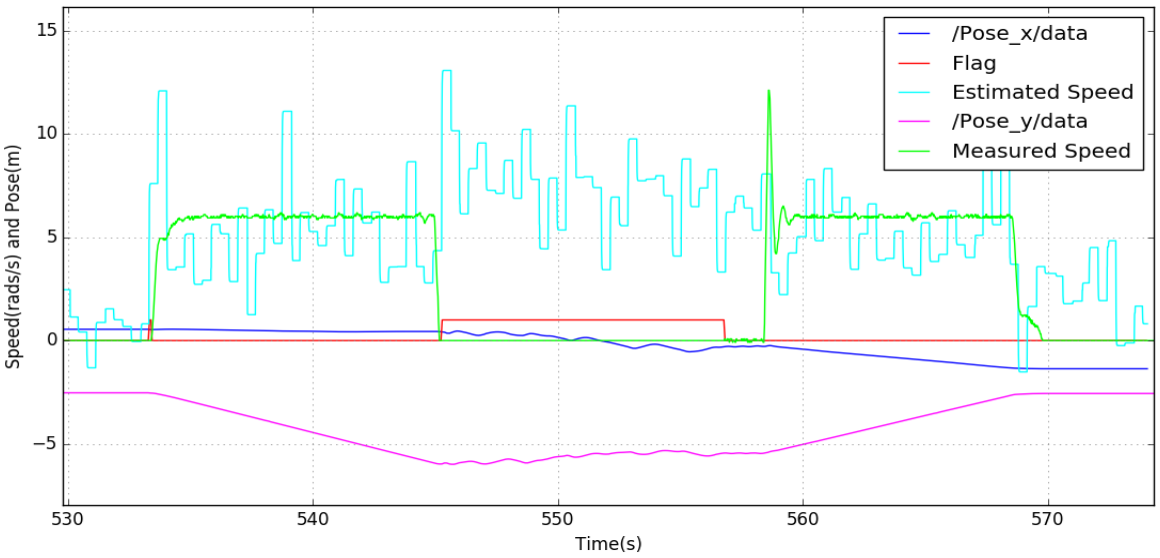


Figure 29 the Result of Localisation Task with Fault Tolerant Control (Estimated Speed by Measured Current

An alternative estimation is using the full mathematical motor model and the estimated armature voltage from the duty cycle control signals. Except for the estimation in the model node, all of the other nodes are the same as those in the experiment of Figure 29. Figure 30 and Figure 31 show the results. Similarly, as Figure 30 shows, when the wires are pulled out, the measured speed (pink) from the encoder on the right wheel suddenly drops to zero. After a small delay, the flag is on. During this process, the control signal (green) is unchanged, as well as the estimated speed.

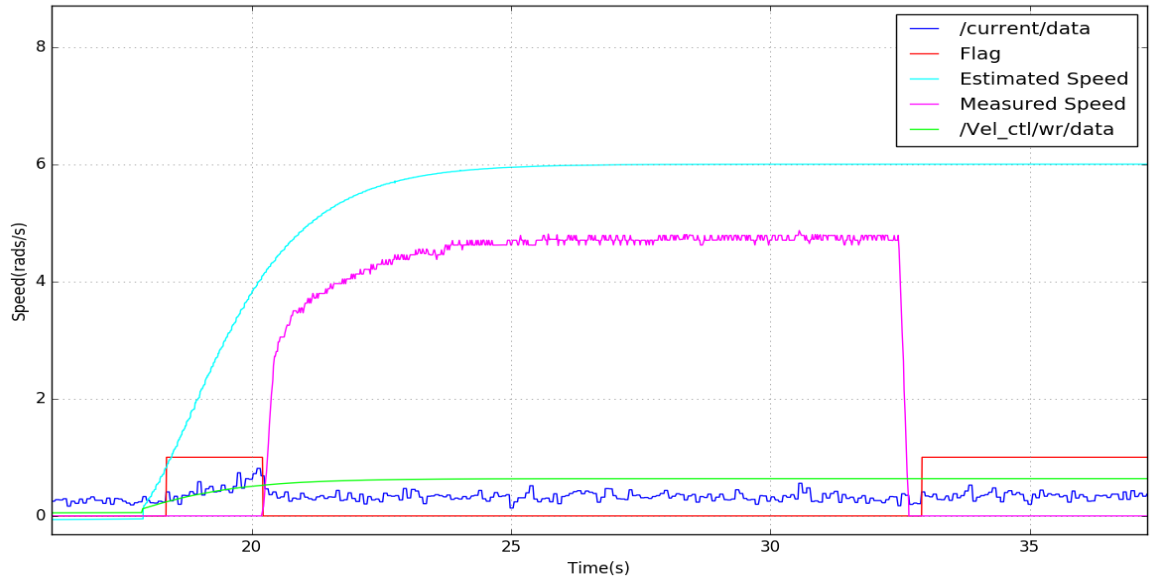


Figure 30 Control Signal, Flag Signal, Estimated Speed, and Measured Speed when Fault is Made (Estimated Speed by Estimated Voltage)

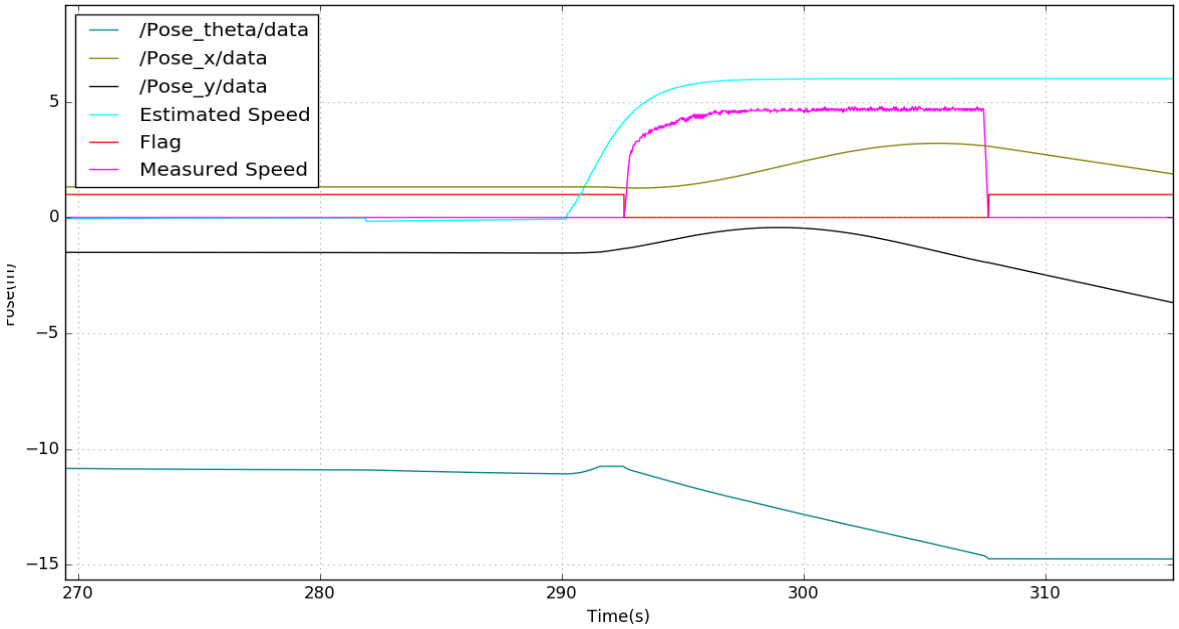


Figure 31 Localisation Task with Fault Tolerant Control (Estimated Speed by Estimated Voltage)

Accordingly, the pose result was then tested as shown in Figure 31. The estimated and measured speeds have the same colours in those in Figure 4.2. While the flag is on, the pose on the y-axis (black) keeps the same direction as the pose when the flag is off. The direction of the robot on the x-axis as well as theta slightly changes at the moment when the flag is on. However, the changed stops then. It could be observed that the increase of the angle theta stops after that the flag is on.

5. Discussions

This project involves many steps in order to increase the reliability of the fault tolerant control. Besides some logical reasons that influence the process of experiments are explained in the methodology part, there are other vital characteristics need to be discussed and analysed.

5.1 Characteristics of the Motor Driver

The original signal of the armature voltage in the motor circuit is a periodic square wave with a magnitude of battery voltage, while the plot data was tested by the multimeter which demonstrates RMS values of the voltages.

5.2 Characteristics of the Motor

5.2.1 Relations among the Speed, Voltage and Current of the Motor

The current from the current sensor as well as the torque should be zero when the input armature input is zero. The Equation (2.5.5) becomes $V = IR$. Suppose that the armature voltage is gradually increased from zero. At first, the motor would not rotate because of inertia, while the torque is increasing. Since V is non-zero, the armature current I is, therefore, non-zero, which indicates that the torque is non-zero. In this case, the armature torque will increase with the current with increasing V. This explains why the currents are larger than those under other duty cycle ratios when the ratio is from 0.2 to 0.3.

According to Equation (2.5.2) and (2.5.3), when the motor starts rotating, the back electromotive force increases from zero and is proportional to the speed. Meanwhile, the armature current will be reduced from a large value to a stable low value which is still larger than zero. If the speed is increased, it is the armature voltage rather than the current that increases with speed for directly providing the counter electromotive force. This is the reason that the current stays constant when the duty cycle ratio is increased during the rotation.

5.2.2 Parameters Tuning

As for the mathematical model, values of the parameters influence the output speed with a step duty cycle percentage input. Regarding the motor source voltage as the input, and the angular speed as the output, applying the Laplace transform, the transfer function is as equation (5.2.1).

$$H(s) = \frac{\acute{\omega}(s)}{V(s)} = \frac{K}{(Js+b)(Ls+R)+K^2} = \frac{K/JL}{s^2+\frac{JR+bL}{JL}s+\frac{K^2}{JL}} \tag{5.2.1}$$

According to the characteristics of a second order transfer function $\frac{g}{s^2+2\varepsilon ws+w^2}$, the response speed depends on the natural frequency $w = \sqrt{\frac{K^2}{JL}}$. Correspondingly, increasing J and L, or decreasing K, might decrease the natural frequency to slow the response. On the

other hand, the damping properties are influenced by the damping coefficient $\varepsilon = \frac{\frac{JR+bL}{JL}}{2\sqrt{\frac{K^2}{JL}}} = \sqrt{\frac{(JR+bL)^2}{2K^2JL}}$. The smaller the damping coefficient, the oscillatory the speed response might be.

Table 4 Changes on Damping Coefficient and Natural Frequency with Increasing Parameters

Increasing Parameters	Damping Coefficient ε	Natural Frequency w
Inertia J	Increase	Decrease
Armature Resistance R	Increase	-
Damping of the Mechanical System b	Increase	-
Armature Induction L	Increase	Decrease
Electromotive Constant K	Decrease	Increase

Comparing with the experiment results in Section 3.2.2, however, the trending of performance changing does not entirely match that in theory, which indicates that the mathematical model might not be identical to the real motor. Comparing reality with the model, decreasing k indicates that the transfer efficiency from electrical to mechanical is smaller; increasing b means the damping of the motor such as friction is larger. The actual parameters need to be identified by other approaches.

5.2.3 Currents for Fault Detection

At the start of measuring currents by the current sensor, it is inevitable that the offset voltage from the sensor is not always precisely 2.5 V. Consequently, the offset current is not zero. The appropriate solution is to adjust the initial value in Equation (3.1.1) before every test.

There is a direct relation between the output torque and the armature current referring to Equation (2.5.2). If additional torque is needed, the current will increase. For a single wheel, three types of situations may increase the torque as well as the armature current: the wheel is starting from the still, accelerating, and on a slope. When the wheel is on a slope, the measured speed from is non-zero, and the current may increase. Therefore, there are two types of situations that the measured speed from the encoder is zero, whilst the armature current is much larger than $0.25\ A$ referring Figure 22. One situation is that the duty cycle is

non-zero but too small to rotate the motor as shown in Figure 24; another is that the encoder is broken.

In this case, the situation of the broken encoder includes that the LED light in the encoder is off, the disk is damaged, or the peripheral wires are disconnected. These faults on encoder are irreducible that needs to be repaired by the human. In general, these two situations are counted as faults that to be detected.

Besides, the induction in the armature circuit may lead to delays. In some cases, when a duty cycle ratio is given, the current may not immediately increase, as Figure 4.2 shows. Therefore, there is a delay between the control signal and the measured speed. When the duty cycle ratio is set to be zero, the current may not drop quickly. Instead, the time of decreasing for the current is considerable. These situations may lead to the measured speed is zero, while the current is significant, which may cause unreliable fault detections.

5.3 The Reliabilities of Fault Tolerant Control for Localisation

For the estimation from the measured current, shown in Figure 29, the reliability is relatively low. Theoretically, the noises of the measured currents are Gaussian. When calculating the pose by estimated speed from the measured currents, the process of integration may cancel most of the noises. Additionally, the moving average filter is added on the current to improve the current. However, the results show that these noises still affect the performance. Another severe problem is that if the motor is already rotating, the current might not increase with the duty cycle ratio. Therefore, the reliability of the estimated speed directly from the measured current is unsatisfactory.

Another type of estimation from the estimated armature voltage, shown in Figure 30 and Figure 31, does not have Gaussian noises, could bring smooth results. There are also several problems limiting the reliability. First, the data on the output voltages from the motor driver under different duty cycle ratios was tested by the multimeter, and approximated by a polynomial equation. Second, the input of the estimation is the control signal of the duty cycle ratio. Due to the induction of the motor armature circuit, usually, there is a delay between this control signal and the current, as well as the motor rotational speed. The outcome is that the estimated speed usually leads the measured speed, which probably causes unreliable fault detection.

Another potential problem is because of the PID speed control. Considering a set-point velocity for driving straight line is input, the fault tolerant logic in PID node is that when a flag is detected, the estimated speed replaces the measured speed. If P parameter is high in order to achieve excellent performance, at the moment of pulling out the wires on the encoder, the measured speed from the encoder immediately drops to zero, while the flag still needs a short delay time to be on, so the measured speed has not been replaced yet. The control force will quickly increase the duty cycle control signal. This problem will cause the robot to lose control to drive in a circle rather than to drive straight.

If the P parameter is low, the setting time of acceleration will increase, while the estimated speed immediately increases with the duty cycle signal. Therefore, the measured speed will lag the estimation, which may cause unreliable fault detection.

6. Conclusions and Future work

6.1 Conclusions

In this dissertation, the problem of fault-tolerant control on mobile robots is addressed. The main contribution is the implementation of the fault tolerant control on the Puzzle Bot. While the robot is executing the localisation task, and the encoder is broken, the fault can be detected, and the estimated data can automatically replace the measurements to continue the task.

An essential discussion on the characteristics of the motor driver and the DC motor is presented. The analysis on the intrinsic features of the motor such as inertia, hysteresis, and induction plays an essential role in improving the performance of the mathematical motor model for estimation. Due to the inappropriate parameters from the reference, there are also contributions to improving the reliability of the estimation such as improving the filter and tuning the parameters. Discovering the relation between the RMS value of the armature voltage and duty cycle ratio is another contribution to improvements. By analysing and concluding those characteristics, the behaviours of encoder fault are also distinguished, in order to make proper fault detection logic. Situations on changes of currents, voltages, and speeds, are all discussed.

Another contribution relies on the comparison of two types of estimation of the motor rotational speeds by different source data, and application on the localisation task. Meantime, the reliabilities of these two estimations are also analysed.

6.2 Future Work

Due to lack of time, many detailed experiments have been left for the future work. Future work is mainly on further improving the reliability of the estimations in existing experiment steps, and the availability of the fault tolerant control. There are some directions that need to be concerned.

1. It would be meaningful to improve the reliability of the source data of the estimation. In this project, the estimated armature voltages and the measured currents are both used as the source data. The former one is lack of direct data support (it is measured by multimeter and estimated by the trending line), while the latter is easily influenced by many intrinsic features of the motor. It would be interesting to explore a new type of estimation like choosing inertial instrument unit as another type of sensor, or improve the reliability of these two existing source data. To improve the existing two, the alternative approach is using more reliable measuring instruments such as using oscilloscope to test the armature voltages and currents.
2. The working principle of the motor is complex that there are characteristics, such as hysteresis, induction, which are nearly unpredictable, are difficult to measure. In this case, it is recommended to perform more reliable tests that could conclude the features and reduce the bias as much as possible.
3. Considering the process of tuning the parameters of the mathematical motor model, it is essential to strictly identify the tuned parameters are the real parameters of the physical motor. There are approaches on DC motor parameter identification such as using speed step response, measuring by multimeter, and simulation methods. It is advised to find out the parameter one by one and then identify the system as a whole.
4. During the tests the delays caused by hysteresis and induction in the motor are inevitable, while the fault detection logic could be improved to avoid the effect on

localisation performance by the delays as much as possible. The PID parameters are also expected to be well-tuned.

5. Concerning the performance of localisation, it is also expected to extend to perform other different localisation tasks, as well as detect other types of faults. For example, the fault on motor is expected to detected and identified. Meanwhile, it is important to distinguish the situations of different faults.

7. Bibliography

[1] L. Allegros MicroSystems, “Fully Integrated, Hall Effect-Based Linear Current Sensor IC with 2.1 kVRMS Isolation and a Low-Resistance Current Conductor,” 2006.

[2] A. Huges, Electric Motors and Drives, Elsevier Ltd., 2006.

[3] Blanke, M., Kinnaert, M., Lunze, J., Staroswiecki, M., Diagnosis and Fault-Tolerant Control, Springer, 2016.

[4] Peter J. Brockwell, Richard A. Davis, Introduction to Time Series and Forecasting, Second Edition, Springer, 2002.

[5] Yilin Zhao, S.L. BeMent, “Kinematics, dynamics and control of wheeled mobile robots,” in *Proceedings 1992 IEEE International Conference on Robotics and Automation*, Nice, France, 1992.

[6] P. B. R. Isermann, “Trends in the application of model-based fault detection and diagnosis of technical processe,” *Control Engineering Practice,* vol. 5, no. 5, pp. 709-719, 1997.

[7] Youmin Zhang,Jin Jiang, “Bibliographical review on reconfigurable fault-tolerant control systems,” in *Annual Reviews in Control*, 2008.

[8] S. W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing, San Diego, CA: California Technical Publishing, 1999.

[9] Neenu Thomas, Dr. P. Poongodi, “Position Control of DC Motor Using Genetic Algorithm Based PID Ctroller,” in *the World Congress on Engineering*, London, 2009.

[10] Mordechai Ben-Ari, Francesco Mondada, Elements of Robotics, Springer, 2018.

[11] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, Andrew Ng, “ROS: an open-source Robot Operating System,” in *ICRA Workshop on Open Source Software*, 2009.

[12] B. Willey, “ROS.org,” Open Source Robotics Foundation, 10 07 2017. [Online]. Available: http://wiki.ros.org/roscpp/Overview/Time. [Accessed 01 09 2018].

[13] Mogens Blanke, W.Christian Frei, Franta Kraus, J.Ron Patton, Marcel Staroswiecki, “What is Fault-Tolerant Control?,” in *IFAC Proceedings Volumes*, Budapest, Hungary, 2000.

[14] Mohd Azri Abdul Aziz, Mohd Nasir Taib, Ramli Adnan, “Moment of inertia of a DC motor as significant factor on the performance of PSO algorithm utilizing WTRI based fitness function,” in *IEEE Conference on Systems, Process and Control (ICSPC)*, Bandar Hilir, Malaysia, 2016.

[15] Roland Siegwart, Illah R. Nourbakhsh, Introduction to Autonomous Mobile Robots, Massachusetts Institute of Technology, 2004.

8. Appendix

8.1 Current Sensor Characteristic Test

According to the datasheet, when the load circuit is off, the output voltage should be 2.5V. The current could be worked out by Equation (2.3.2). Based on the datasheet [12], which states that the noise of the voltage is within 21mV, the noise of the current should be within

$$\frac{21mV}{185mV/A} = 0.114A \tag{8.1.1}$$

To test the characteristics of the current sensor, a simple circuit was built as Figure 4.1.1 with a 51Ohm resistor and a 5V power source.

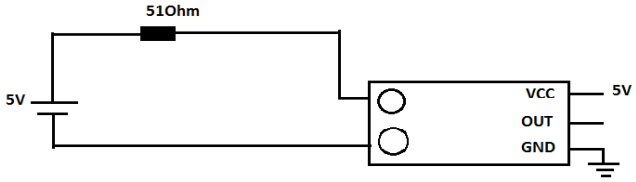


Figure 3.2.1 Schematic Presentation of the Current Sensor Test

The oscilloscope shows that the output voltage changed from $2.435V$ to$2.445V$. Theoretical current value should be

$$\Delta V = 0.01V \rightarrow I = \frac{0.01V}{0.185V/A} = 0.054A \tag{8.1.2}$$

And the current value could also be calculated by testing the resistance and amp meter:

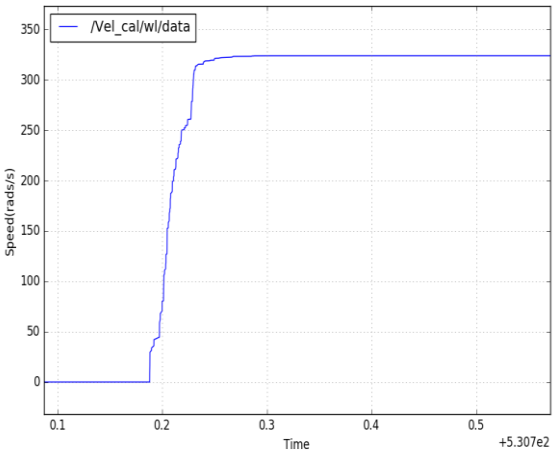
$$I = \frac{4.8V}{51Ohm} = 0.09A \tag{8.1.3}$$

This is within $0.114A$. It proves that the current sensor is under normal working condition.

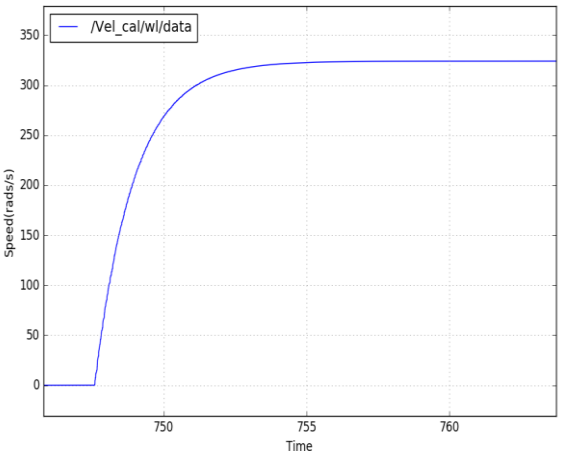
8.2 Mathematical Motor Model Parameter Test

Results are shown in Figure 9.2.1. Case 1 is the recommended set of parameters, which has a setting time within $0.1s$, and no overshoots. The steady state value of Case 1 is around $330\,rads/s$. In Case 2, only the inertia J is increased. The setting time of Case 2 is over $7s$. The steady state value is same as the value of Case 1, and the overshoot is also zero. In Case 3, b is increased to $3.5e^{-4}\ Nms$. The steady state value decreases to around $90\ \ rads/s$, and the setting time is still within $0.1s$, without overshoots.

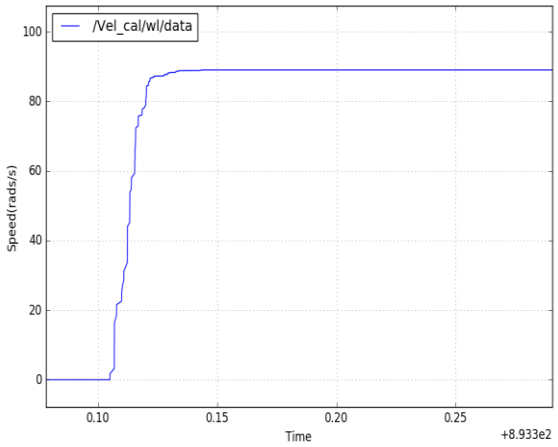
In Case 4, L is increased from $0.003H$ to $0.3H$. The steady state value is still at around 330 $rads/s$, while there is an overshoot over 30%. In Case 5, K is increased from 0.03334 Nm/Amp to 0.3334 Nm/Amp. The number of damping increases much more. The largest overshoot is 270%, while the steady state value dramatically decreases from 330 $rads/s$ to about 35 $rads/s$. In Case 6, R is reduced from $8.7\ \Omega$ to $1.7\ \Omega$. The steady state value is at 330 $rads/s$. The settling time is slightly increased since there is a small overshoot about 6%.



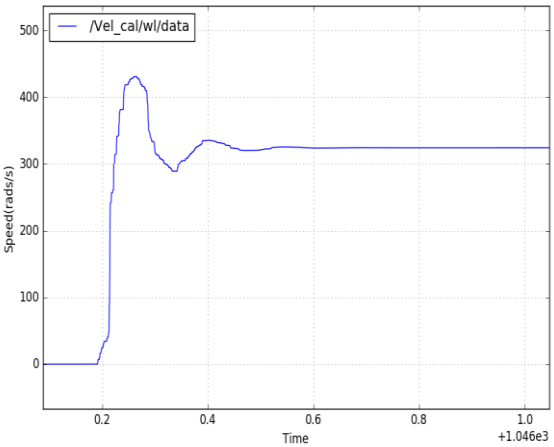
Case 1



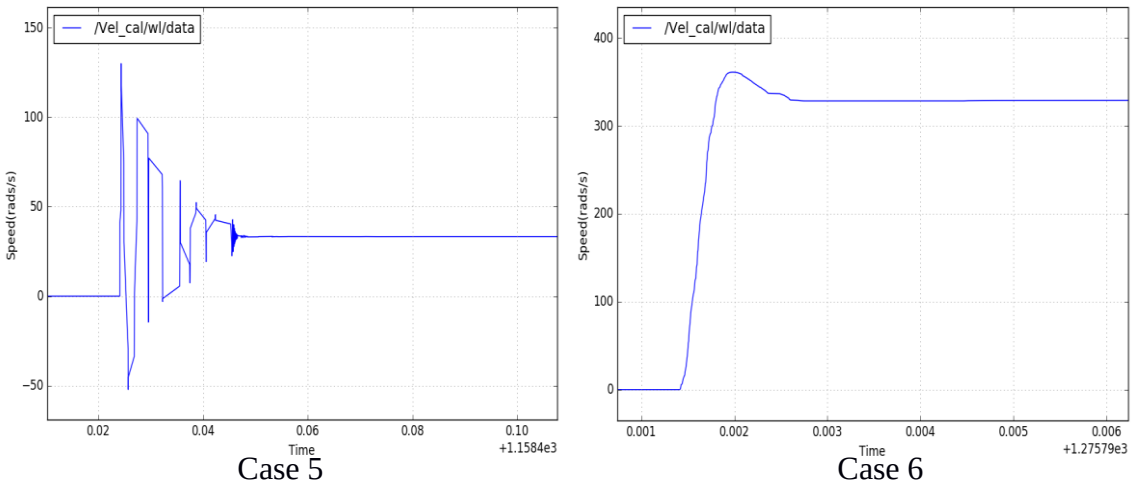
Case 2



Case 3



Case 4



Case 5 Case 6

Figure 9.2.1 Experiment Results of Estimated Speed with Sets of Parameters in Table 3.1

8.3 PID Speed Control

The node PID is used to calculate the corresponding duty cycle control signal by an input reference speed. The nodes $/feedback$ and $/serial\_node$ are the same with those in the previous experiments in Section 3.2.2.

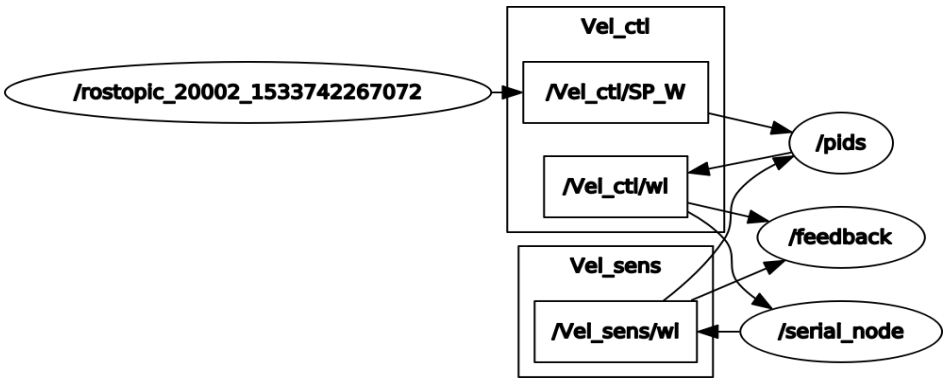


Figure 8.3.1 Node Graph when the nodes Arduino, Feedback, PID added

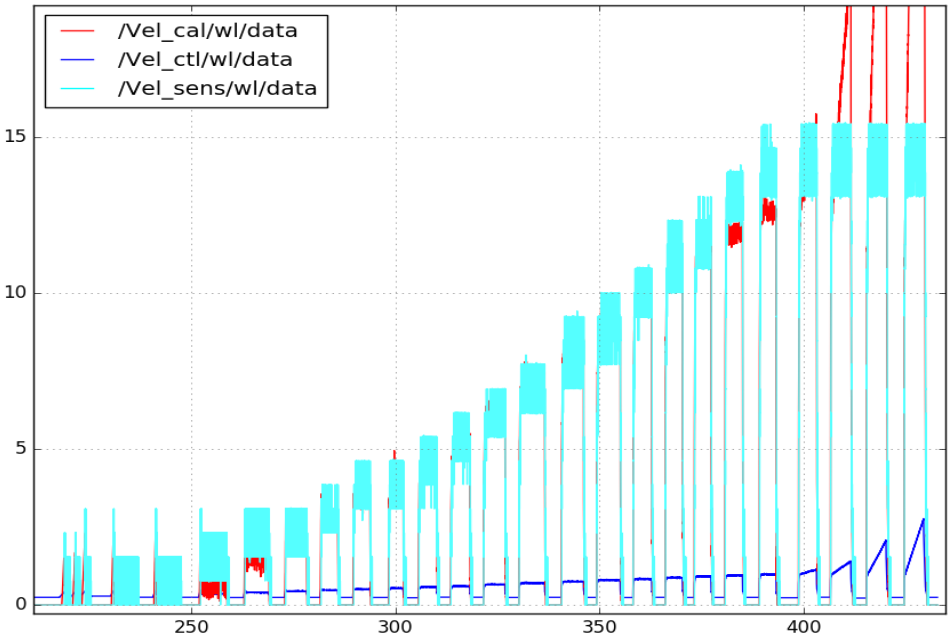


Figure 8.3.2 Speeds Control with PID

8.4 Puzzle Robot Localisation Test

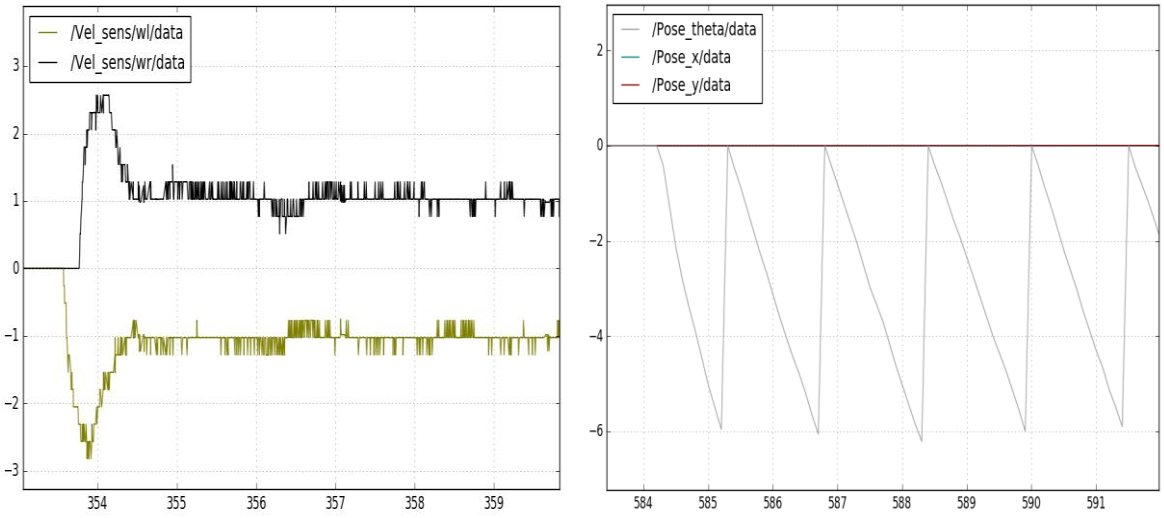


Figure 8.4.1 Localisation Test on Circle Trajectory

8.5 Main Codes

Feedback.cpp

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
#include "std_msgs/Int32.h"
#include "iostream"
#include "math.h"
```

```
using namespace std;

class MotorSensor
{
        volatile int buff;
        volatile double w;
        volatile double ave,crt_p;
        double s_wl,dt,current;
        double R,k1,k2,J,L,b,m,u_l,u;
        double i_now,i_prev,w_now,w_prev;
        double rads_l;
        double motor_current_l;
      double ctr_w;
        ros::NodeHandle node;
      ros::Subscriber sens_vel_wl,subcrt,subvtg,rot_vel_wl;
        ros::Subscriber u_wl;
      ros::Publisher i_cal_wl,Vel_cal_wl,curet;
      ros::Time current_time, last_time;
public:
        MotorSensor();
        void Run();
        void WlSenscallback(const std_msgs::Float32& msg);
        void crt_lSenscallback(const std_msgs::Float32::ConstPtr& msg);
        void vtg_lSenscallback(const std_msgs::Float32& msg);
        void controlvl(const std_msgs::Float32::ConstPtr& msg);
//      void ucallback(const std_msgs::Float32& msg);

};

MotorSensor::MotorSensor(){

        Vel_cal_wl = node.advertise<std_msgs::Float32>("Vel_cal/wl",100);
        i_cal_wl   = node.advertise<std_msgs::Float32>("i_cal/wl",100);
 //     curet = node.advertise<std_msgs::Float32>("i_sens/wl",10);

        sens_vel_wl=node.subscribe("Vel_sens/wl",100,&MotorSensor::WlSenscallback,this)
;
        subcrt = node.subscribe("current",1000,&MotorSensor::crt_lSenscallback,this);
        subvtg = node.subscribe("voltage",100,&MotorSensor::vtg_lSenscallback,this);
        rot_vel_wl=node.subscribe("Vel_ctl/wl",100,&MotorSensor::controlvl,this);
//      u_wl=node.subscribe("u/wl",1000,&MotorSensor::ucallback,this);

        //R=38;
        R=8.7;
        k1=0.03334;
        k2=0.03334;
        //k1=0.0095;
```

```
        //k2=0.0095;
        //k2=0.01;
        //k1=0.01;
        //J=1.8*(1e-4);
        //J=1.8*(1e-6);
        J=1.8*(1e-6);
        L=0.003;
        b=3.5*(1e-6);
        //b=2.5*(exp(-10));
        //b=3.1*(1e-7);
        //b=1.1*(1e-4);
        //b=2.0*(1e-4);
        m=0.00000;
        u_l=0;
        s_wl=0;
        i_now=0;
        i_prev=0;
        w_now=0;
        w_prev=0;
        ctr_w=0;
        current=0;
        buff=0;
        ave=0;
        w=0;
        //dt=0.001;
current_time = ros::Time::now();
   last_time = ros::Time::now();
}

//void MotorSensor::ucallback(const std_msgs::Float32& msg)
//{
//   u_l = msg.data;
//}
void MotorSensor::controlvl(const std_msgs::Float32::ConstPtr& msg)
{
   ctr_w = msg->data;

//    if(i_now<0)
//        i_now=0;

}

void MotorSensor::WlSenscallback(const std_msgs::Float32& msg)
{

   ROS_INFO("W_l = %g", msg.data);
```

```
}

void MotorSensor::crt_lSenscallback(const std_msgs::Float32::ConstPtr& msg)
{
  //ROS_INFO("current_l = %g", msg.data);
  //i_prev = msg.data;
  current = msg->data;


  ROS_INFO("i_now = %g", i_prev);

 // ROS_INFO("w_now = %g", w_now);
}

void MotorSensor::vtg_lSenscallback(const std_msgs::Float32& msg)
{
  ROS_INFO("vtg_l = %g",msg.data);

}

void MotorSensor::Run(){
    double dt;
    current_time = ros::Time::now();
    dt = (current_time - last_time).toSec();
    last_time = current_time;

 //   u_l = ctr_w*0.645 - 0.186;
  //  u_l=11.1;
   // u_l=0;
  //  if(ctr_w>1)
//        {
//        ctr_w=1;
//        }
    //u_l=2.5*ctr_w+1.63;
 //   u_l = ctr_w * 0.66 - 0.215;
 //   if(u_l<0)
//        {
//        u_l=0;
//        }

 //   i_now=dt*(-R*i_prev-k1*w_prev+u_l)/L+i_prev;
 //   i_now=i_now*0.2+0.8*i_prev;
 //   i_prev = i_now;
 //   i_now = current*0.3 + i_prev*0.7;
 //   i_now = (i_now-0.03)/20;
```

```
 //   if(i_now<0){
//        i_now=0;}

 //   crt_w = current*0.3 + 0.7 *crt_p;
  //  i_now=current-0.25;
  //  w_now = dt*(k2*i_prev-b*w_prev-m)/J+w_prev;
 //   u_l = ctr_w * 0.66 - 0.215;
    //u_l = ctr_w * 11.1;
 //   u_l = 10.325 * pow(ctr_w,2.0)+0.8846*ctr_w -0.1229;
    i_now=dt*(-R*i_prev-k1*w_prev+11.1)/L+i_prev;
    w_now=dt*(k2*i_now-b*w_prev-m)/J+w_prev;
    i_prev=i_now;
    w_prev=w_now;
 //   w=w+w_now;
 //   buff++;
  //  if(buff>2000)
//        {buff=0;ave=w/2000;w=w_now;}

   // w_prev=w_prev+(w_now-400);
 //   if(w_now<0){
 //     w_now = 0;}
    ROS_INFO("i_now = %g", i_now);
    ROS_INFO("current = %g", current);
    //ROS_INFO("w_now = %g", (1e-2));
   // if(w_now>50){
//        w_now=0;}

//left calculated current
    std_msgs::Float32 i_wl;
    i_wl.data = i_now;
    i_cal_wl.publish(i_wl);


//left calculated speed
    std_msgs::Float32 w_wl;
    w_wl.data = w_now;
    Vel_cal_wl.publish(w_wl);
//current
 //   std_msgs::Float32 cs_wl;
 //   cs_wl.data=current;
  //  curet.publish(cs_wl);


}


int main(int argc, char **argv)
```

```
{

 cout<<"Initialising MotorSensor connection"<<endl;

 ros::init(argc, argv, "feedback");

 cout<<"MotorSensor conncted"<<endl;

 MotorSensor motorsensor;

 ros::Rate loop_rate(10000);

 while(ros::ok())
 {
  ros::spinOnce();

  motorsensor.Run();

  loop_rate.sleep();
 }

 return 0;
}
```

2motor.cpp

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
#include "std_msgs/Int32.h"
#include "iostream"
#include "math.h"

using namespace std;

class ModelRos
{
   float s_wr,s_wl,dt,current;
   double R,k1,k2,J,L,b,m,u_l,u_r;
   double i_l_now,i_l_prev,w_l_now,w_l_prev,i_r_now,i_r_prev,w_r_now,w_r_prev;
   double rads_l,rads_r;
   double ctr_w_l, ctr_w_r;
   ros::NodeHandle node;
   ros::Subscriber r_vel_wl, r_vel_wr, sens_vel_wr,sens_vel_wl,subcrt, subvtg;
   ros::Publisher Vel_i_wl, Vel_i_wr,i_cal_wl ;
   ros::Time current_time, last_time;
public:
   ModelRos();
```

```
  void Run();
//   void crt_lSenscallback(const std_msgs::Float32& msg);
//   void crt_rSenscallback(const std_msgs::Float32& msg);
  void WlctlCallback(const std_msgs::Float32::ConstPtr& msg);
  void WrctlCallback(const std_msgs::Float32::ConstPtr& msg);
  void crt_lSenscallback(const std_msgs::Float32::ConstPtr& msg);
  void vtg_lSenscallback(const std_msgs::Float32& msg);

};

ModelRos::ModelRos(){
  // sens_vel_wl=node.subscribe("Vel_sens/wl",100,&MotorSensor::WlSenscallback,this);
  // sens_vel_wr=node.subscribe("Vel_sens/wr",100,&MotorSensor::WlSenscallback,this);
   r_vel_wl=node.subscribe("Vel_ctl/wl",100,&ModelRos::WlctlCallback,this);
   r_vel_wr=node.subscribe("Vel_ctl/wr",100,&ModelRos::WrctlCallback,this);
   subcrt = node.subscribe("current",1000,&ModelRos::crt_lSenscallback,this);
   subvtg = node.subscribe("voltage",100,&ModelRos::vtg_lSenscallback,this);
//    i_cal_wl   = node.advertise<std_msgs::Float32>("i_cal/wl",100);
//    Vel_i_wl = node.advertise<std_msgs::Float32>("Vel_cal/wl",10);
   Vel_i_wr = node.advertise<std_msgs::Float32>("Vel_cal/wr",10);

//   subcrt_l = node.subscribe("current",100,&MotorSensor::crt_lSenscallback,this);
//   subcrt_r = node.subscribe("current",100,&MotorSensor::crt_rSenscallback,this);

        //Initialize variables
   //R=8.7;
   R=38;
   k1=0.01;
   k2=0.01;
   J=1.8*(1e-6);
   L=0.003;
   b=2.0*(1e-4);
   //b=3.1*(exp(-7));
   m=0;
   s_wr=0;
   s_wl=0;
   ctr_w_r=0;
   u_r=0;


   i_l_now=0;
   i_l_prev=0.0;
   w_l_now=0;
   w_l_prev=0.0;

   i_r_now=0;
   i_r_prev=0.0;
```

```
   w_r_now=0;
   w_r_prev=0.0;
   current=0;


   current_time = ros::Time::now();
   last_time = ros::Time::now();
   //dt=0.001;
}


          //Callback functions


//void FeedbackRos::crt_lSenscallback(const std_msgs::Float32& msg)
//{
//      current = msg->data;
//}
//void FeedbackRos::crt_rSenscallback(const std_msgs::Float32& msg)
//{
//      current = msg->data;
//}

void ModelRos::WlctlCallback(const std_msgs::Float32::ConstPtr& msg)
{
          ctr_w_l = msg->data;
}

void ModelRos::WrctlCallback(const std_msgs::Float32::ConstPtr& msg)
{
          ctr_w_r = msg->data;
}

void ModelRos::crt_lSenscallback(const std_msgs::Float32::ConstPtr& msg)
{
          current = msg->data;
}
void ModelRos::vtg_lSenscallback(const std_msgs::Float32& msg)
{
  ROS_INFO("vtg_l = %g",msg.data);

}

void ModelRos::Run(){
   double dt;
   current_time = ros::Time::now();
   dt = (current_time - last_time).toSec();
```

```
   last_time = current_time;

//left speed estimated

//    u_l = 10.325 * pow(ctr_w_l,2.0)+0.8846*ctr_w_l -0.1229;
//    i_l_now=dt*(-R*i_l_prev-k1*w_l_prev+u_l)/L+i_l_prev;
//    w_l_now=dt*(k2*i_l_prev-b*w_l_prev-m)/J+w_l_prev;
//    i_l_prev=i_l_now;
//    w_l_prev=w_l_now;
//    if(w_l_now<0){
//       w_l_now = 0;}

//right speed estimated

   u_r = 10.325 * pow(ctr_w_r,2.0)+0.8846*ctr_w_r -0.1229;

   i_r_now=dt*(-R*i_r_prev-k1*w_r_prev+u_r)/L+i_r_prev;
 //   i_r_prev = current - 0.25;
   w_r_now=dt*(k2*i_r_prev-b*w_r_prev-m)/J+w_r_prev;
   i_r_prev=i_r_now;
   w_r_prev=w_r_now;
//    if(w_r_now<0){
//       w_r_now = 0;}

//left current sensor calculated speed
//    std_msgs::Float32 i_wl;
//    i_wl.data = w_l_now;
//    Vel_i_wl.publish(i_wl);
//right current sensor calculated speed
   std_msgs::Float32 i_wr;
   i_wr.data = w_r_now;
   Vel_i_wr.publish(i_wr);
}

int main(int argc, char **argv)
{

  cout<<"Initialising FeedbackRos connection"<<endl;

  ros::init(argc, argv, "motor");

  cout<<"FeedbackRos connected"<<endl;

  ModelRos modelros;

  ros::Rate loop_rate(10000);
```

```
 while(ros::ok())
 {
  ros::spinOnce();

  modelros.Run();

  loop_rate.sleep();
 }

 return 0;
}
```

check_fault.cpp

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
#include "std_msgs/Int32.h"

using namespace std;

class CheckRos
{
	double w_sens_wl,w_sens_wr,w_cal_wl,w_cal_wr;
	double threshold;
	double current;
	int flag_l, flag_r;
	ros::NodeHandle node;
	ros::Publisher vel_flag_wl, vel_flag_wr;
	ros::Subscriber sens_vel_wl, sens_vel_wr, cal_vel_wl, cal_vel_wr,subcrt;


public:
	CheckRos();
	void Run();
	void WrSensCallback(const std_msgs::Float32::ConstPtr& msg);
//	void WlSensCallback(const std_msgs::Float32::ConstPtr& msg);
	void WrcalCallback(const std_msgs::Float32::ConstPtr& msg);
//	void WlcalCallback(const std_msgs::Float32::ConstPtr& msg);
      void crt_lSenscallback(const std_msgs::Float32::ConstPtr& msg);

};

CheckRos::CheckRos()
{
	//publisher
//	vel_flag_wl = node.advertise<std_msgs::Float32>("Vel_warn/wl",10);
	vel_flag_wr = node.advertise<std_msgs::Float32>("Vel_warn/wr",10);
```

```
	//subscriber
	sens_vel_wr=node.subscribe("Vel_sens/wr",10,&CheckRos::WrSensCallback,this);
//	sens_vel_wl=node.subscribe("Vel_sens/wl",10,&CheckRos::WlSensCallback,this);

//	cal_vel_wr =node.subscribe("Vel_cal/wr",10,&CheckRos::WrcalCallback,this);
//	cal_vel_wl =node.subscribe("Vel_cal/wl",10,&CheckRos::WlcalCallback,this);
	subcrt = node.subscribe("current",1000,&CheckRos::crt_lSenscallback,this);

	threshold = 0.4;
//	flag_l=0;
	flag_r=0;
	current=0;
}


void CheckRos::WrSensCallback(const std_msgs::Float32::ConstPtr& msg)
{
		w_sens_wr = msg->data;
}
//void CheckRos::WlSensCallback(const std_msgs::Float32::ConstPtr& msg)
//{
//		w_sens_wl = msg->data;
//}
//void CheckRos::WrcalCallback(const std_msgs::Float32::ConstPtr& msg)
//{
//		w_cal_wr = msg->data;
//}
//void CheckRos::WlcalCallback(const std_msgs::Float32::ConstPtr& msg)
//{
//		w_cal_wl = msg->data;
//}
void CheckRos::crt_lSenscallback(const std_msgs::Float32::ConstPtr& msg)
{
		current = msg->data;
}

void CheckRos::Run()
{
//	if(!((w_sens_wl <(w_cal_wl+threshold))&&((w_cal_wl-threshold)<w_sens_wl)))
//		flag_l=1;
//	else
//		flag_l=0;
//	if(!((w_sens_wr <(w_cal_wr+threshold))&&((w_cal_wr-threshold)<w_sens_wr)))
//		flag_r=1;
//	else
//		flag_r=0;
```

```
        if(w_sens_wr == 0)
        {
                if((current > 0.4))
                        {
                                flag_r=1.0;
                        }
        }
        else
        {
                flag_r=0.0;
        }

//      std_msgs::Float32 flagl;
//      flagl.data = flag_l;
//      vel_flag_wl.publish(flagl);

        std_msgs::Float32 flagr;
        flagr.data = flag_r;
        vel_flag_wr.publish(flagr);

}

int main(int argc, char **argv)
{
        cout<<"checking faults"<<endl;
        ros::init(argc, argv, "check");
        cout<<"Check_Ros conncected"<<endl;
        CheckRos check_ros;
        ros::Rate loop_rate(100);
        while(ros::ok())
        {
                ros::spinOnce();
                check_ros.Run();
                loop_rate.sleep();
        }
        return 0;



}
```

pose.cpp

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
```

```
#include "std_msgs/Int32.h"
#include "iostream"
#include "fstream"
#include "math.h"


using namespace std;


class PoseRos
{	double vr, wr,w_r;
	double w_cal_wl,w_cal_wr;
	float w_sens_wr,w_sens_wl;
	int flag_r;
	double x, y, theta, x_dot, y_dot, theta_dot;

	ros::NodeHandle node;
	ros::Subscriber
sens_vel_wr,sens_vel_wl,cal_vel_wr,cal_vel_wl,vel_flag_wl,vel_flag_wr;
	ros::Publisher  ref_vel_V,ref_vel_W,pose_x,pose_y,pose_theta,sens_wr;
	ros::Time current_time, last_time;

public:
	PoseRos();
	void Run();
	void WrSensCallback(const std_msgs::Float32::ConstPtr& msg);
	void WlSensCallback(const std_msgs::Float32::ConstPtr& msg);
	void WrcalCallback(const std_msgs::Float32::ConstPtr& msg);
//	void WlcalCallback(const std_msgs::Float32::ConstPtr& msg);
//	void WlflagCallback(const std_msgs::Float32::ConstPtr& msg);
	void WrflagCallback(const std_msgs::Float32::ConstPtr& msg);

};

PoseRos::PoseRos()
{

//	ref_vel_V = node.advertise<std_msgs::Float32>("Vel_ctl/SP_V",10);
//	ref_vel_W = node.advertise<std_msgs::Float32>("Vel_ctl/SP_W",10);
	pose_x = node.advertise<std_msgs::Float32>("Pose_x",10);
	pose_y = node.advertise<std_msgs::Float32>("Pose_y",10);
	pose_theta = node.advertise<std_msgs::Float32>("Pose_theta",10);
//	sens_wr = node.advertise<std_msgs::Float32>("/Vel_sens/wr",10);

	sens_vel_wr=node.subscribe("Vel_sens/wr",20,&PoseRos::WrSensCallback,this);
	sens_vel_wl=node.subscribe("Vel_sens/wl",20,&PoseRos::WlSensCallback,this);
	cal_vel_wr =node.subscribe("Vel_cal/wr",20,&PoseRos::WrcalCallback,this);
```

```
//        cal_vel_wl =node.subscribe("Vel_cal/wl",20,&PoseRos::WlcalCallback,this);

//        vel_flag_wl = node.subscribe("Vel_warn/wl",20,&PoseRos::WlflagCallback,this);
          vel_flag_wr = node.subscribe("Vel_warn/wr",20,&PoseRos::WrflagCallback,this);

          x=0;
          y=0;
          theta=0;
          x_dot=0;
          y_dot=0;
          theta_dot=0;
          vr=0;
          //wr=0;
          w_r=0;
          flag_r=0;
          w_cal_wl=0;
          w_cal_wr=0;
          w_sens_wr=0;
          w_sens_wl=0;

}


void PoseRos::WrSensCallback(const std_msgs::Float32::ConstPtr& msg)
{
          w_sens_wr = msg->data;
}

void PoseRos::WlSensCallback(const std_msgs::Float32::ConstPtr& msg)
{
          w_sens_wl = msg->data;
}
void PoseRos::WrcalCallback(const std_msgs::Float32::ConstPtr& msg)
{
          w_cal_wr = msg->data;
}
//void PoseRos::WlcalCallback(const std_msgs::Float32::ConstPtr& msg)
//{
//        w_cal_wl = msg->data;
//}

void PoseRos::WrflagCallback(const std_msgs::Float32::ConstPtr& msg)
{
          flag_r = msg->data;
}

void PoseRos::Run()
```

```
{
	double dt;
	float radius=0.046;
	float l = 0.255;
	//calculate dt;
	current_time = ros::Time::now();
	dt = (current_time-last_time).toSec();
	last_time = current_time;

	if(flag_r == 1)
	{
		w_r=w_cal_wr;

//		std_msgs::Float32 fake_wr;
//		fake_wr.data = w_cal_wr;
//		sens_wr.publish(fake_wr);
	}
	else
	{
		w_r=w_sens_wr;
	}

	//wr = (w_r-w_sens_wl)/l;
	vr = radius*(w_r+w_sens_wl)/2;
	//changing rate

	x_dot = vr*cos(theta);
	y_dot = vr*sin(theta);
	theta_dot=radius*(w_r-w_sens_wl)/l;

	//pose
	x = x + dt * x_dot;
	y = y + dt * y_dot;
	theta= theta + dt * theta_dot;
//	if(theta > 2*3.14159265358979323946)
//		theta=0;
//	if(theta < -2*3.14159265358979323946)
//		theta=0;

	std_msgs::Float32 p_x;
	p_x.data = x;
	pose_x.publish(p_x);

	std_msgs::Float32 p_y;
	p_y.data = y;
	pose_y.publish(p_y);
```

```
        std_msgs::Float32 p_theta;
        p_theta.data = theta;
        pose_theta.publish(p_theta);

//      std_msgs::Float32 v_c;
//      v_c.data = 0.08;
//      ref_vel_V.publish(v_c);

//      std_msgs::Float32 w_c;
//      w_c.data = 0.5;
//      ref_vel_W.publish(w_c);

}


int main(int argc, char **argv)
{
        cout<<"Initialising Pose Setting" << endl;
        ros::init(argc,argv,"pose");
        PoseRos pose_ros;


        ros::Rate loop_rate(20);
        while(ros::ok())
        {
        ros::spinOnce();

        pose_ros.Run();

        loop_rate.sleep();
        }
        return 0;
}
```

1motor_sensor.ino

```
#if (ARDUINO >= 100)
 #include <Arduino.h>
#else
 #include <WProgram.h>
#endif

#include <Servo.h>
#include <ros.h>
#include <std_msgs/Float32.h>
```

```
#include <std_msgs/Int32.h>
#include <geometry_msgs/Pose.h>
#include<tf/tf.h>
double wl;

//#include<Filter.h>
//ExponentialFilter<long> ADCFilter(10,0);


//Define ports for Left encoder
#define LeftEncoderInterruptA 0
#define LeftEncoderInterruptB 1


//set up variables foe left encoder
volatile bool _LeftEncoderASet;
volatile bool _LeftEncoderBSet;
volatile bool _LeftEncoderAPrev;
volatile bool _LeftEncoderBPrev;
volatile long _LeftEncoderTicks = 0;
long left_encoder_vec[100];
long time_vector[100];
float current_amp[20];

//only  set it up once
volatile int tcnt2 = 236;
//volatile int tcnt2 = 99;
int vel_window,buf_size=100;
long enc_prev,count_enc=0;
int buffer_size=20;
volatile int count_crt=0;

float D_S_l;
float D_S_r;
float x=0,y=0,theta=0;
volatile float sum=0;
float dt;

//Time
unsigned long current_time, previous_time;
long t=0,prev_micros = 0;

// ros message set up
ros::NodeHandle  nh;
std_msgs::Float32 Wl_msg;
std_msgs::Int32 counter_left;
std_msgs::Float32 current;
```

```
std_msgs::Float32 voltage;


//ros publishers setup
ros::Publisher W_l("Vel_sens/wl", &Wl_msg);
ros::Publisher cnt_left("Count_left", &counter_left);
ros::Publisher crt("current",&current);
ros::Publisher vtg("voltage",&voltage);

int value = 0;
//float smooth = 0;
//float x_new = 0;
float i_new ;
float i_pre ;
float y_new;
float y_prev;


//Servo Initialization
Servo servo;

//Subscriber Callback Functions left motor
void left_vel( const std_msgs::Float32& cmd_msg2){
  double wl=(saturation(cmd_msg2.data)*(-500))+1500;
  //servo2.write(cmd_msg2.data); //set servo angle, should be from 0-180
  servo.writeMicroseconds(wl);
}

//Main Function ROS
////////////////////////////////////////////////////////////////////////////////////////////////
/////////

void read_sensors(){
float  Left_vrad_l=0;
float dt;
long  Left_Nticks;

  if(count_enc<buf_size)
  {

    left_encoder_vec[count_enc]  = _LeftEncoderTicks;
    time_vector[count_enc]= micros();
    count_enc++;
  }
  else
  {
```

```
   for(int i=0;i<buf_size-1;i++)
     left_encoder_vec[i] = left_encoder_vec[i+1];
   for(int i=0;i<buf_size-1;i++)
     time_vector[i] = time_vector[i+1];


   left_encoder_vec[buf_size-1]  = _LeftEncoderTicks;
//    Serial.println(_LeftEncoderTicks);
   time_vector[buf_size-1] = micros();
  }

  Left_Nticks = left_encoder_vec[count_enc-1] - left_encoder_vec[count_enc-2];
  D_S_l=((float)Left_Nticks/400)*(M_PI*0.1016);
  x+=(D_S_r+D_S_l)/2*cos(theta);
  y+=(D_S_r+D_S_l)/2*sin(theta);
  theta+=(D_S_r-D_S_l)/0.365;
  enc_prev = count_enc - vel_window - 1;
  if(enc_prev<0)
    enc_prev = 0;

  Left_Nticks = left_encoder_vec[count_enc-1] - left_encoder_vec[enc_prev];

  dt=(float)(time_vector[count_enc-1] - time_vector[enc_prev])/1e+6;
  if(dt<0 || dt>1)
    dt = 0.15;

  Left_vrad_l=(((float)Left_Nticks*2*M_PI)/400)/dt; //velocity in radians per second

//Publish encoder readings
  Wl_msg.data=(Left_vrad_l);
  W_l.publish(&Wl_msg);
}


void read_current(){
  float crtvalue;
  float vtgvalue;
  value = analogRead(A0);
  vtgvalue = float(value)/1024*5;
  y_new = vtgvalue;
  //y_new = 0.1*y_new+0.9*y_prev;
  voltage.data = vtgvalue;
  crtvalue = (abs(y_new - 2.53))/0.185;
  y_prev = y_new;
  if(count_crt<buffer_size)
  {
```

```
   current_amp[count_crt]  = float(crtvalue);
   sum += current_amp[count_crt];
   count_crt++;
  }
  else if(count_crt==buffer_size)
  {
   i_new = sum/buffer_size;
   sum=0;
   count_crt=1;
   current_amp[count_crt]=crtvalue;
   sum += current_amp[count_crt];
   count_crt++;
  }

//  i_new = crtvalue;
//  i_new = 0.2*i_new + 0.8*i_pre;
  i_pre = i_new;
  current.data = i_pre;
  //current.data = crtvalue;
  crt.publish(&current);
  vtg.publish(&voltage);
}
//buffer and filter
 // value = analogRead(A0);
//  vtgvalue = float(value)/1024*5;
 // voltage.data = vtgvalue;
//  crtvalue= (abs(vtgvalue - 2.4375))/0.185;
 // i_new = crtvalue;
//  if(count_crt<buffer_size)
//  {

//    current_amp[count_crt]  = float(crtvalue);
//    sum += current_amp[count_crt];
//    count_crt++;
//  }
 // else if(count_crt==buffer_size)
 // {
 //   i_new = sum/buffer_size;
 //   sum=0;
 //   count_crt=1;
  //  current_amp[count_crt]=crtvalue;
  //  sum += current_amp[count_crt];
  //  count_crt++;
 // }
//exponential filter
 //  ADCFilter.Filter(value);
```

```
 //  smooth = ADCFilter.Current();
//   x_new = value;
//   i_new = 0.2*i_new+0.8*i_pre;

//Publishers


  // voltage.data = float(value)/1024*5;
 //  i_pre = i_new;
 //  current.data = i_pre;
  // current.data = (abs(voltage.data - 2.4375))/0.185;
 //  crt.publish(&current);
 //  vtg.publish(&voltage);

//}

//Subscribers and callback functions
   ros::Subscriber<std_msgs::Float32> sub2("Vel_ctl/wl", &left_vel);

//MAIN LOOP
void setup()
{
//Initialize ROS node handler (subscribe and advertise topics)
  nh.initNode();
//  Serial.begin(57600);
  nh.subscribe(sub2);

  nh.advertise(W_l);

  nh.advertise(cnt_left);
  nh.advertise(crt);
  nh.advertise(vtg);
  //Serial.begin(9600);


//Set up pins for H-Bridge
  servo.attach(10); //attach it to pin 10 for arduino mega

// Port Configuration for encoders
  DDRD = DDRD | 0x00;
  PORTD = 0b00001111;


//Set Interrupts for encoders -Left
  attachInterrupt(digitalPinToInterrupt(21), HandleLeftMotorInterruptA, CHANGE);
  attachInterrupt(digitalPinToInterrupt(20), HandleLeftMotorInterruptB, CHANGE);
```

```
  //Set interrupts for timer - only once!
  TIMSK2 &= ~(1<<TOIE2);
  TCCR2A &= ~((1<<WGM21) | (1<<WGM20));
  TCCR2B &= ~(1<<WGM22);
  TCCR2B |= (1<<CS21);
  TCCR2B &= ~(1<<CS20);
  TCCR2B &= ~(1<<CS22);
  ASSR &= ~(1<<AS2);
  TIMSK2 &= ~((1<<OCIE2A) | ~(1<<OCIE2B));
  TCNT2 = tcnt2; // defined as 130
  TIMSK2 |= (1<<TOIE2);

//Initialize Variables
  vel_window = 10;
  left_encoder_vec[0] = 0;
  time_vector[0]=0;
  //current_time=micros();
  //previous_time=micros();
}

//MAIN PROGRAM
void loop()
{
 // float crtvalue;
 // float vtgvalue;
 // value = analogRead(A0);
  //vtgvalue = float(value)/1024*5;
  //voltage.data = vtgvalue;
 // crtvalue = (abs(vtgvalue - 2.5))/0.185;
 // i_new = crtvalue;
 // i_new = 0.2*i_new + 0.8*i_pre;
 // i_pre = i_new;
 // current.data = i_pre;
 // crt.publish(&current);
//  vtg.publish(&voltage);
  read_current();
  nh.spinOnce();
  read_sensors();
 // read_current();

//  wl=1500;
//  servo.writeMicroseconds(wl);
  delay(1);
}
```

```
// Interrupt service routines for the left motor's quadrature encoder
   //Channel A
void HandleLeftMotorInterruptA(){
  _LeftEncoderBSet = PIND & (1<<PD0);
  _LeftEncoderASet = PIND & (1<<PD1);

  _LeftEncoderTicks+=ParseEncoder_L();

  _LeftEncoderAPrev = _LeftEncoderASet;
  _LeftEncoderBPrev = _LeftEncoderBSet;
}

   //Channel B
void HandleLeftMotorInterruptB(){
  _LeftEncoderBSet = PIND & (1<<PD0);
  _LeftEncoderASet = PIND & (1<<PD1);

  _LeftEncoderTicks+=ParseEncoder_L();

  _LeftEncoderAPrev = _LeftEncoderASet;
  _LeftEncoderBPrev = _LeftEncoderBSet;
}



// Logic for left encoder count
long ParseEncoder_L(){
  if(_LeftEncoderAPrev && _LeftEncoderBPrev){
    if(!_LeftEncoderASet && _LeftEncoderBSet) return 1;
    if(_LeftEncoderASet && !_LeftEncoderBSet) return -1;
  }else if(!_LeftEncoderAPrev && _LeftEncoderBPrev){
    if(!_LeftEncoderASet && !_LeftEncoderBSet) return 1;
    if(_LeftEncoderASet && _LeftEncoderBSet) return -1;
  }else if(!_LeftEncoderAPrev && !_LeftEncoderBPrev){
    if(_LeftEncoderASet && !_LeftEncoderBSet) return 1;
    if(!_LeftEncoderASet && _LeftEncoderBSet) return -1;
  }else if(_LeftEncoderAPrev && !_LeftEncoderBPrev){
    if(_LeftEncoderASet && _LeftEncoderBSet) return 1;
    if(!_LeftEncoderASet && !_LeftEncoderBSet) return -1;
  }
  return 0;
}


//Saturation function (x>1 or x<-1)
```

```
float saturation (float x){
  if (x>1)
    return 1;
    else if (x<-1)
    return -1;
    else
    return x;
}

//Timer Function
ISR(TIMER2_OVF_vect) {

}
```

Launch.launch

```
<launch>
        <node name="feedback" pkg="feedback" type="feedback"/>
<!--    <node name="pids" pkg="feedback" type="pids"/>  -->
<!--    <node name="pose" pkg="feedback" type="pose"/>  -->
        <node pkg="rosserial_python" type="serial_node.py" name="serial_node">
                <param name="port" type="string"  value="/dev/ttyACM0"/>
                <param name="baud" type="int"     value="57600"/>
        </node>
<!--    <node name="motor" pkg="feedback" type="motor"/>   -->
</launch>
```