Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

ROS2 Project: Robotic Car

Robotics 814
Report 2: Moving With Control Using ROS

Dylan Charl Eksteen
22623906

March 28, 2023

Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, Matieland 7602, South Africa.

Plagiarism declaration

I have read and understand the Stellenbosch University Policy on Plagiarism and the definitions of plagiarism and self-plagiarism contained in the Policy [Plagiarism: The use of the ideas or material of others without acknowledgement, or the re-use of one's own previously evaluated or published material without acknowledgement or indication thereof (self-plagiarism or text-recycling)].

I also understand that direct translations are plagiarism, unless accompanied by an appropriate acknowledgement of the source. I also know that verbatim copy that has not been explicitly indicated as such, is plagiarism.

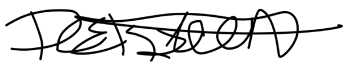
I know that plagiarism is a punishable offence and may be referred to the University's Central Disciplinary Committee (CDC) who has the authority to expel me for such an offence.

I know that plagiarism is harmful for the academic environment and that it has a negative impact on any profession.

Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully (acknowledged); further, all verbatim copies have been expressly indicated as such (e.g. through quotation marks) and the sources are cited fully.

I declare that, except where a source has been cited, the work contained in this assignment is my own work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment. I declare that have not allowed, and will not allow, anyone to use my work (in paper, graphics, electronic, verbal or any other format) with the intention of passing it off as his/her own work.

I know that a mark of zero may be awarded to assignments with plagiarism and also that no opportunity be given to submit an improved assignment.

Signature: 

Name: DC Eksteen Student no: 22623906

Date: 28/03/2023

Table of contents

List of figures

Assignment Overview

This report is for the completion of the ROS2 project for Robotics 814. The project aimed to use ROS2 and python to make a robotic car drive in an s-shape.

This is the second report as part of the project and covers the setting up of the ROS nodes that will handle the control of the car, as well as the feedback-control that was implemented.

Submission Requirements

- A video of the robot moving.
- The code used for the project
- A short report that lists and explains key parts of the code.
- The ROS Computation Graph.

ROS Implementation

For the application of this project three ROS2 nodes were created and used. An *imu_node* that collects and publishes the IMU data to an *imu_data* topic, a *wheels_node* that subscribes to an *wheels_data* topic and controlls the PWM signal that is sent to each wheel's motor and finally a *controller_node* that subscribes to the *imu_data* topic, infers some reference angle and applies the control method by publishing an adjustment factor to the *wheels_data* topic. These nodes and topics are shown in Figure 1 below.

Figure 1: ROS Graph of Implemented Nodes and Topics

The nodes were then run in their own operating threads by launching them from different terminals. This is demonstrated in Figure 2 below, where each terminal will host it's own node.

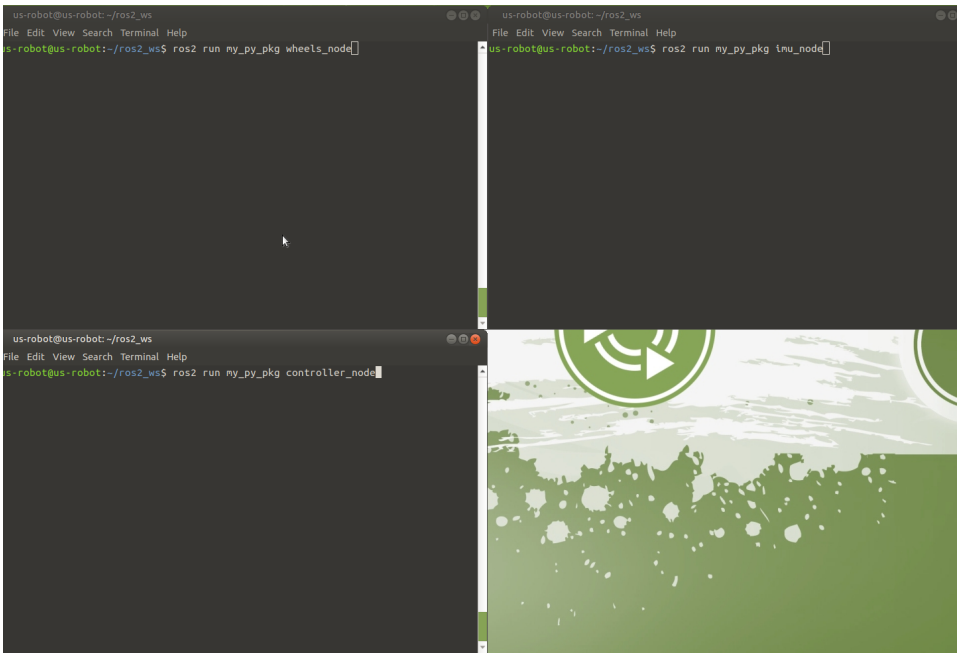


Figure 2: Terminals Before Starting Nodes

This application of the nodes utilises one of the main advantages of using ROS: the nodes can operate independently. Figure 3 shows the three nodes after the *controller_node* has completed its control loop and the car has again stopped. As can be seen in the figure, the *wheels_node* has received a stop message and is now waiting for a start message to start the wheels turning again and the *imu_node* is still publishing the IMU data to the *imu_data* topic.

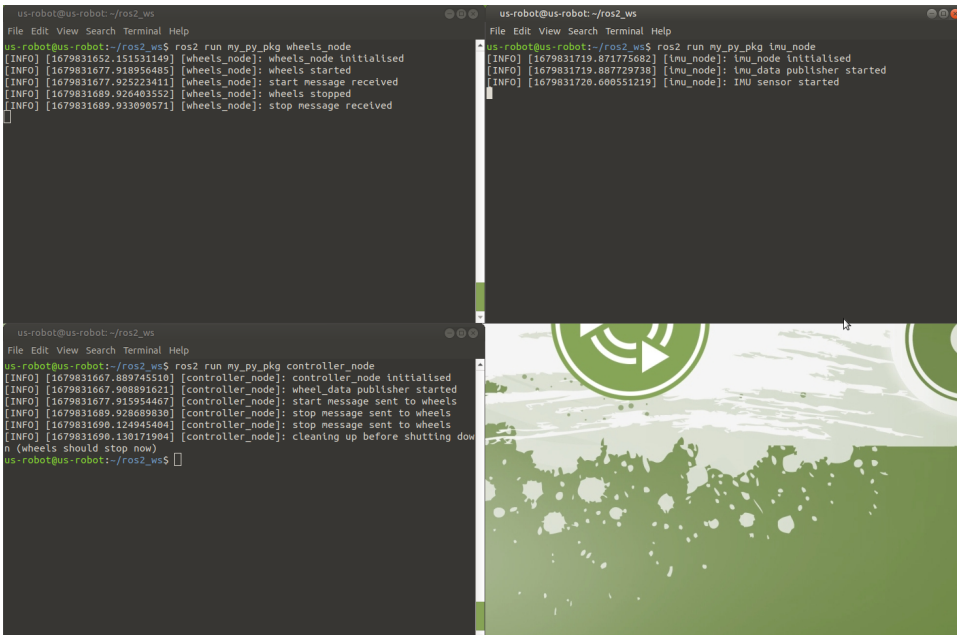


Figure 3: Terminals After Control Node has Terminated

Topic Descriptions

The two topics that were implemented for the project were named *imu_data* and *wheel_data* respectively.

imu_data

The *imu_node* publishes the z-axis angular acceleration in degrees/s at a frequency of 100 Hz. This transmission continues independently of the action of other nodes and topics until the *imu_node* is terminated. The data is published as a *String* message from the built-in ROS example messages and converted to a float value after it is received by the *controller_node*.

wheel_data

The *controller_node* publishes a PWM correction factor on this topic that should be added to the right wheel's PWM value and subtracted from the left wheel's PWM value. Further, the *controller_node* will publish the term "start" and "stop" respectively when it requires the wheels to start or stop moving. The messages are all published as a *String* message from the built-in ROS example messages and converted to a float value after it is received by the *wheels_node*.

Software Overview

As was mentioned in Report 1, All the code for the project was written in Python and interpreted using Python 3 interpreter. All of the information and instructions for the ROS part of the project are from an online course that was taken as part of the module. (Renard, 2023)

IMU_Node Overview

The *imu_node* is responsible for reading the imu data from the mpu_6050 imu module mounted to the bottom of the car. The node has all of the functions required to initialize the module and convert the data to the required format and units built in.

The code for the node was written in an object-oriented style where all of the functions and operations of the node are contained within a single *mpu_9250* class.

The following code snippet gives an overview of the methods defined as part of the class:

```
class mpu_9250(Node):

    def __init__(self): ...
    def start_publisher(self): ...
    def imu_publisher(self): ...
    def mpu6050_start(self): ...
    def read_raw_bits(self, register): ...
    def mpu6050_conv(self): ...
```

When the *mpu_9250* class is finished initializing, a timer is started that calls the *imu_publisher()* method to publish the data from the IMU sensor to the *imu_data* topic.

Within the *__init__()* method, all of the register addresses and offsets are initialized as can be found in the data sheet for the mpu_6050 IMU module. These registers data are then read with the method call *read_raw_bits()*, where the register argument specifies the register that is to be read from. The options here are the three axis measurements of the gyrometer, or the three axis measurements of the accelorometer.

In this case, all of the data is read from the registers, as this will allow for easier inclusion of other control methods to the application. In the applied case, the angular velocity (gyrometer) around the z-axis was used for the control, so this is the value that was published to the *imu_data* topic.

Wheels_Node Overview

The *wheels_node* is responsible for the control of the two DC motors connected to the wheels of the car via the Raspberry Pi GPIO pins. All of the GPIO and PWM controls are contained and controlled from within this node.

The node subscribes to the *wheels_data* topic where it receives the PWM adjustment values from the *controller_node*. The data is also checked to see when a "start" or "stop" command is received so that the node can start or stop the wheels turning.

The code for this node was also written using an object-oriented programming approach, and the following code snippet shows an overview of the *WheelsNode* class that defines the behaviour of the node.

```
class WheelsNode(Node):

    def __init__(self): ...
    def start_subscriber(self): ...
    def wheel_data_callback(self, msg): ...
    def start_wheels(self): ...
    def stop_wheels(self): ...
    def wheel_control(self): ...
    def sigint_handeler(self, signum, frame): ...
    def cleanup(self): ...
```

One notable feature of the *wheels_node* is that it will always run the *cleanup()* method upon termination. This is to ensure that the GPIO pins are properly cleared in order to ensure reliable utilization of the pins in future programs.

Controller_Node Overview

The *controller_node* is responsible for the main control of the behaviour of the car. The node subscribes to the *imu_data* topic and saves the IMU sensor data to be used in a control equation. The node then publishes the control signal to the *wheels_data* node where the PWM signal sent controlling the car's motors is altered.

The node is also responsible for the timing of all of the important functions of the car and implements a timer to trigger the control calculation and publishing to the *wheels_data* topic. The node is also responsible for collecting and saving the data from the IMU sensor for analysis of the control implementation.

The code for the *controller_node* was implemented using an object-oriented programming approach, and the methods that are defined within the *Controller* class can be seen in the code snippet below:

```
class Controller(Node):

    def __init__(self): ...
    def start_program(self): ...
    def stop_program(self): ...
    def start_subscriber(self): ...
    def imu_data_callback(self, msg): ...
    def start_publisher(self): ...
    def sigint_handeler(self, signum, frame): ...
    def wheel_data_publisher(self): ...
    def cleanup(self): ...
    def start_wheels(self): ...
    def stop_wheels(self): ...
    def control(self): ...
    def save_data(self): ...
    def PID_output(self): ...
```

When the node is terminated, or the program finishes, a “stop” message is sent to the *wheels_node* causing the wheels to stop, then the collected data is saved to a CSV file and the node destroys itself freeing up the system resources for further use, while the other nodes are still running and can merely continue operating if another instance if the *controller_node* is run.

Applied Control Algorithm

Proportional, Integral and Derivative control (PID) was implemented for this assignment. PID control determines the behaviour of the system according to Equation 1 below.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{1}$$

where:

$u(t)$ is the control signal or output,

$e(t)$ is the error signal or difference between the desired set point and the actual output at time t,

K_p, K_i, and K_d are the proportional, integral, and derivative gain constants, respectively.

Since the data from the IMU sensor tends to be very noisy, and the data is received at a frequency of 100 Hz, a windowed average of the angular velocity is used rather than the instantaneous measurement. This smoothes out the measured data and results in a more reliable control implementation. The effect of this smoothing is seen in Figure 4 below.

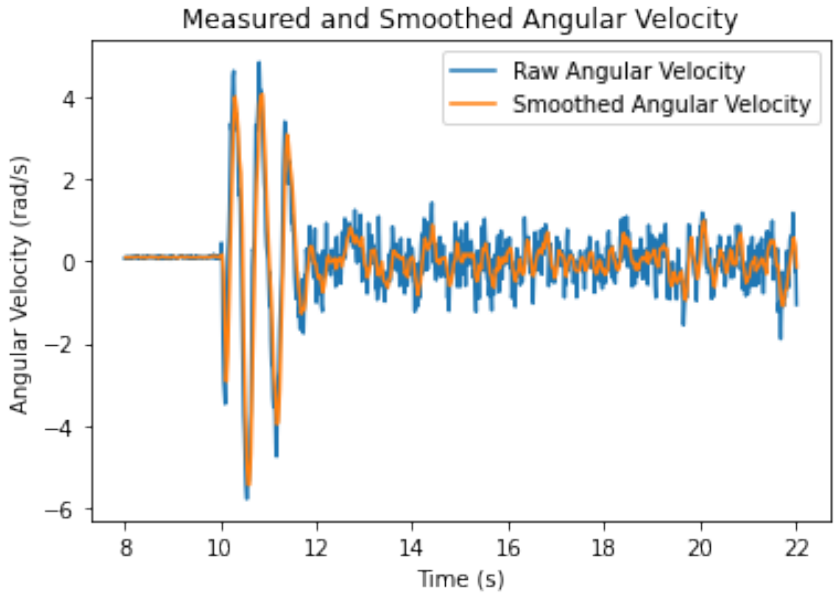


Figure 4: Raw Measured and Smoothed Sensor Data

Constant Reference Angle

In testing, the first test to determine the effectiveness of the control algorithm was to give a constant reference angle and to determine if the car stays in a straight line. Figure 5 below shows the measured response graph of the car when following a constant reference angle.

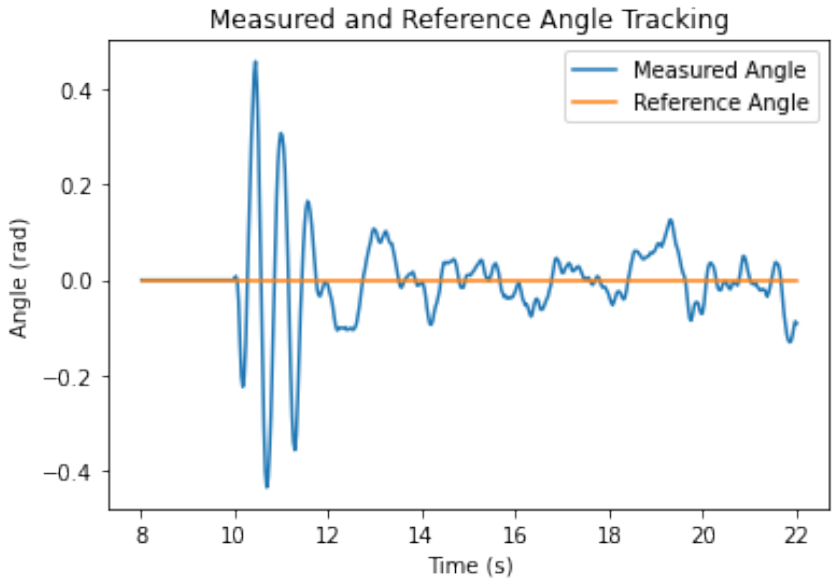


Figure 5: Control Response to Constant Angle Tracking

The car drove almost perfectly straight when the PID control was active, which is a very large improvement over the uncontrolled movement observed in Report 1.

Sin Wave Reference Angle

In order to achieve the assignment instruction of making the car move in an "S-shape", a sin wave was used as a reference angle. After turning the controller gain parameters as well as the reference input frequency and amplitude, the control graph in Figure 6 was achieved.

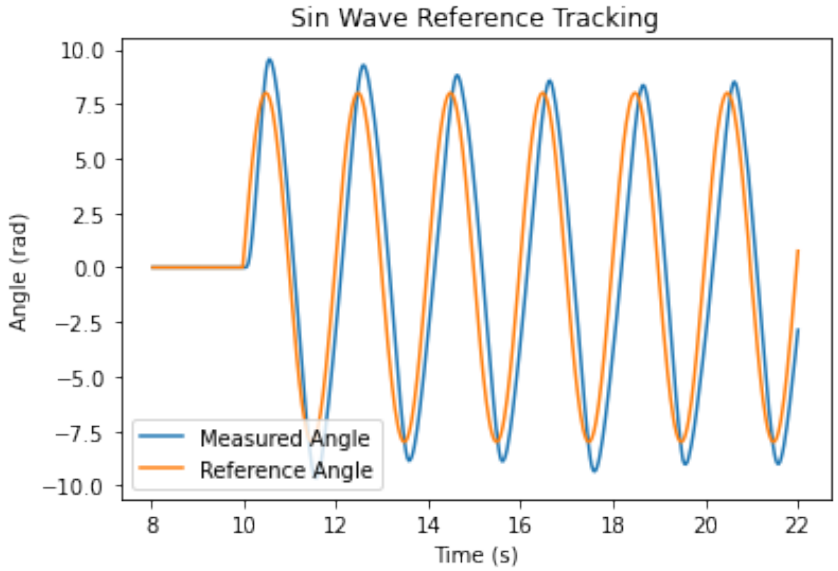


Figure 6: Control Response to Sin Wave Tracking

In this graph, we can see that the car managed to smoothly and accurately track the sin wave reference angle, and the resulting movement was observed to be satisfactory.

Disturbance Rejection

The final test analysis of the implemented control algorithm was to see how well it manages to handle a large external disturbance. In Figure 7 below, we can observe that a large disturbance was applied at approximately 15 seconds. At this time the car was lifted and rotated about 90°.

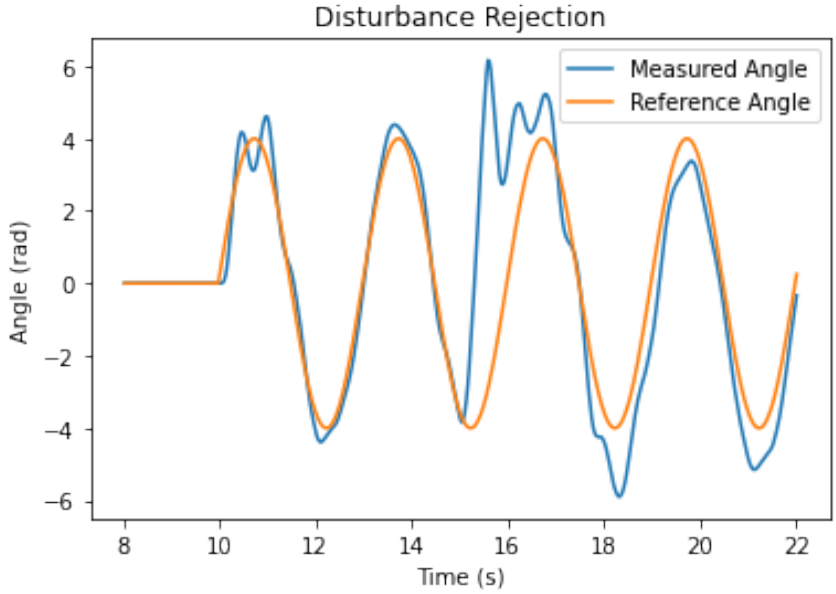


Figure 7: Control Response to External Disturbance

From the figure, it is clear to see that a relatively good disturbance rejection was achieved by the controller as the car soon after the disturbance manages to follow the reference input again. Visually this was also the case, as the car immediately corrected its orientation after being put down and continued on its S-shaped path.

Conclusions

This project set out to build and program a car that can move in an s-shape using ROS2 implemented on a Raspberry Pi. This report demonstrated the application of ROS to the problem and described the nodes and topics that were deployed to achieve the desired functionality.

The report also describes and evaluates the control algorithm that was implemented to control the movement of the car. All of the desired goals for the project were achieved and a lot of knowledge and experience on the subject of ROS2 programming was gained in the process.

All of the code that was implemented during this project can be found at `https://github.com/TheMechatronic/ROS2_Robotic_Car`

List of references

Renard, E. (2023). Ros2 for beginners (ros foxy, humble - 2023) | udemy. Available at: https://www.udemy.com/course/ros2-for-beginners/learn/lecture/21305292?start=135#overview