

Getting MIRTE under (ros2_)control

A thesis on improving the system response of the wheels of the MIRTE Master as observed from `ros2_control`

Master Thesis by

Jasper van Brakel

In order to obtain the degree of Master of Science in Robotics
at Delft University of Technology,
To be defended publicly on Monday February 23, 2026 at 3:00PM.

Student Number: 5323215
Faculty: Mechanical Engineering
Thesis committee: Prof. Dr. ir. Martijn Wisse (Supervisor)
Dr. Micah Prendergast
Dr. Jordan Boyle
Ir. Gijs van der Hoorn

An electronic version of this thesis is available at <http://repository.tudelft.nl>.



Getting MIRTE under (`ros2_`)control

A thesis on improving the system response of the wheels of the MIRTE Master as observed from `ros2_control`

Jasper van Brakel

Supervisor: Prof. dr. ir. Martijn Wisse

16th February 2026

Abstract—A good wheel system response is crucial for accurate robot locomotion, especially when using a Mecanum wheel locomotion system. However, the step and frequency responses of TU Delft’s budget-oriented educational robot, the MIRTE Master, leave much to be desired. The purpose of this thesis is to improve the frequency and step response of the wheels of the MIRTE Master while keeping its budget, hardware, and flexibility constraints intact. This has been achieved by improving the control architecture to better integrate with the `ros2_control` framework and optimizing the hardware communication. This has resulted in a modular control architecture with a controller-frequency independent noise reduction of the wheel velocity signal. Furthermore, the computational load of the control setup has been changed to be significantly reduced in the situation where the robot is idle. This new architecture showed significant improvements compared to the original setup in frequency response and step response measurements.

1 Introduction

For robotics students, it is essential to learn how to apply the theory to a real robot. Such an educational robot needs to have an adequate locomotion system to enable novice users to experiment and configure their planning and control algorithms. The MIRTE Master (see Fig. 1) is such an educational robot [1]. However, it has a subpar system response, which makes it difficult to control. In this thesis, I set out to improve the control architecture of the MIRTE Master educational robot in order to improve the system response of the wheel motors.

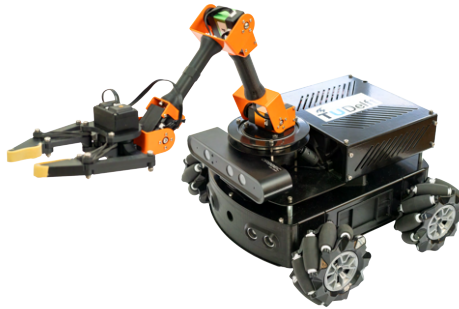
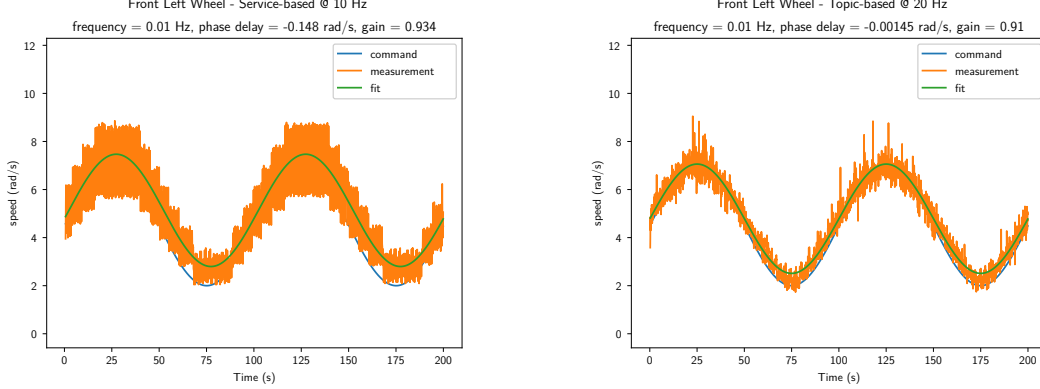


Figure 1: A MIRTE Master robot with Mecanum wheel drive.

The current `ros2_control` implementation for the Mecanum drive base of the MIRTE Master offers a poor step and frequency response, limiting the stable region of the connected controllers. This is demonstrated by the low gain and phase margin of 2.6 dB and 70° of the base open-loop system. The following four causes were identified. First of all, the measured wheel velocity has a lot of noise from the controller’s perspective, as can be seen in Fig. 2a, however this behavior is not observable on the actual system. Second of all, the setup is configured to run at a frequency of 10 Hz, which is low for such a system [2] and far below the limit of the hardware communication setup of 50 Hz.

This limits the maximum reconstructible feedback signal frequency significantly due to the Nyquist-Shanon sampling theorem [3]. Third of all, the wheel velocity command signals are severely discretized in the `ros2_control` hardware interface, while the underlying hardware communication uses a finer quantization resolution, as can also be seen Fig. 2a. Last of all, the current `ros2_control` setup has a significant computational cost, which was the motivation for the use of the extra discretization and the low controller frequency.



(a) The recovered sinusoid of the front left wheel velocity reading of the original `ros2_control` setup running at 10 Hz.

(b) The recovered sinusoid of the front left wheel velocity reading of the proposed `ros2_control` setup running at 20 Hz.

Figure 2: The response of the front left wheel of the original and the proposed setup when driven with an offset sinusoid at 0.01 Hz with offset of 4.5 rad s^{-1} and an amplitude of 2.5 rad s^{-1} .

In this work, an alternative `ros2_control` implementation is proposed in order to alleviate these limitations. This implementation is modular to be more inline with `ros2_control`'s modular hardware interface model. This new design is better suited for tuning to specific hardware instances and is pre-configured to run the base controllers at a frequency of 20 Hz out of the box. Furthermore, the wheel velocity command and state communication is optimized, resulting in less command discretization and fewer messages being exchanged. In addition to this, it calculates the wheel velocity in a frequency-independent way that results in a value closer to the instantaneous wheel velocity with less noise. The combined effects of these changes result in a gain and phase margin of 6 dB and 135° , respectively. This proposed implementation has reduced the system resource utilization when the robot is idle while also experiencing less noise and finer discretization, as illustrated in Fig. 2b.

This thesis is organized as follows; First the relevant historical development background of the MIRTE Master will be described in Section 2. After which, the limitations of the current design will be identified in Section 3. Then the proposed `ros2_control` architecture will be introduced in Section 4. After which, the design and requirements of the performance measurement methods are explained in Section 5. After which, the results of these measurements are described and explained in Section 6. Then a conclusion is drawn, and the findings are summarized in Section 7.

2 Development history of the MIRTE robot series

While this work focuses on the `ros2_control` setup of the MIRTE Master, it is necessary to understand its requirements and history, which have led to the current implementation. Designing, creating, and maintaining a robotics platform is a non-trivial task. Such a process gets more challenging depending on its requirements and target audience. One of the least forgiving situations is as a low-cost educational robot with a broad user base from high school to university level using Robot Operating System (ROS) [4], [5], such as the MIRTE¹ robot series [1].

The first ROS-enabled robot in the MIRTE series was the MIRTE Pioneer (see Fig. 3). This robot is designed for secondary and higher education. It has a layered software architecture that enables the use

¹MIRTE stands for 'MIRTE, an Inspiring Robot for Technology Education'

of the full ROS ecosystem for advanced students while also providing a simplified interface for younger students [1]. The initial design of the MIRTE Pioneer was based on ROS 1 [4] and its ecosystem and conventions, for example, by using `ros_control` to control the differential drive base [6]. At this time, the main objective was to get the MIRTE Pioneer usable for educational purposes, which required a simple-to-configure flexible hardware setup that supported various hardware peripherals.

This flexible hardware setup was achieved by creating a custom ROS hardware communication node called `mirte_telemetrix` or `telemetrix` for brevity. This was based on the Telemetrix project [7] which provides a client-server communication protocol to interface with runtime-configurable microcontrollers. The Telemetrix project’s implementation was used as the base of this work, as it provides several Python server libraries and multiple firmware implementations for various microcontrollers. Over time, the version used by the MIRTE robots received custom extensions and updates to enable more of the required functionality.

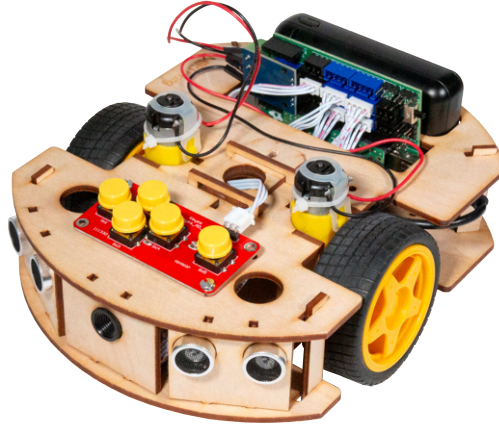


Figure 3: A MIRTE Pioneer, the first and simplest ROS-enabled robot in the MIRTE series.

As the MIRTE Pioneer matured, the more it was attempted to be used in higher education. However, as the standard configuration of the MIRTE Pioneer is intended to be students’ first introduction to robotics, it is lacking capabilities for more complex tasks such as manipulation and advanced perception. The MIRTE Master was designed in order to fulfill this need for a more capable platform (see Fig. 1).

The MIRTE Master is a mobile manipulator-style robot with an omnidirectional Mecanum wheel drive. Furthermore, it is fitted with a more capable single-board computer (SBC), a 2D LiDAR² and a depth camera. Since both robots in the MIRTE series need to be flexible and reconfigurable to aid in their educational goal, it was decided to share as much of the software stack between these robots. This also had the benefit of limiting the amount of software to maintain, while giving the MIRTE Master a head start in development.

At the start, numerous changes needed to be made to the existing MIRTE software stack to enable all the hardware features of the MIRTE Master. Some hardware stability problems emerged when most of these features were added in the ROS 1 version. These issues seemed to be mostly `telemetrix` related, which had trouble keeping up with the messages from the hardware. Some exploration to resolve this was done, and reimplementing `mirte_telemetrix` in C++ looked like a promising solution.

Around this time the last version of ROS 1 was approaching the end of its support window; therefore, it was decided to directly target ROS 2 Humble for this C++ `telemetrix` rewrite. A lot of the work done for the ROS 2 migration, was performed as a one-to-one port. This has left the MIRTE software stack in a peculiar state, as a lot of design patterns and conventions have changed between ROS 1 and ROS 2. Furthermore, ROS (2) Humble is still lacking some features that were available in ROS 1, such as the ability to remap actions. The combination of all this has led to some peculiar designs.

One of these peculiarities is the current `ros2_control` setup for the MIRTE Master, as `ros2_control` has received significant design changes compared to `ros_control` for ROS 1 [6]. This has led to the

²LiDAR stands for ‘light detection and ranging’

initially combined hardware interfaces for the MIRTE Pioneer and Master being split compared to their ROS 1 equivalent; however, significant sections of this code have not been updated to reflect this split or the new `ros2_control` design patterns.

As the MIRTE Master matured, it started to be used more in M.Sc. education and research. This more serious use case has unknowingly increased the (performance) requirements of the MIRTE codebase. This is the result of this user group requiring finer control over their movement and more processing capacity to run their custom work, including, for example, perception, localization, and planning. A significant part of the system resources are being used by the combination of `telemetrix` and `ros2_control`, leaving very little processing time for the other processes, such as other ROS nodes.

Furthermore, the out-of-the-box locomotion performance of the MIRTE Master leaves a lot to be desired. Mecanum wheel drive systems, which are present on the MIRTE Master, are highly sensitive to relative wheel speed differences, as they are used to control the direction of movement [8]–[11]. Consequently, it is critical to perform the commanded wheel velocities accurately and synchronized. These wheel speeds are enforced by a `ros2_control` proportional–integral–derivative (PID) controller, which ships untuned, as a P controller with gain 1, by default as it can differ significantly on a per robot and even on a per motor basis. The system with the PID controller has a rather small usable stable region, in which it has a beneficial effect on the locomotion performance.

3 Problem Identification

For the control architecture and system performance of the MIRTE Master to be improved, the problems with the current system needed to be identified first. This has been done iteratively, initially by user experience stories and manual static code analysis. After which the testing criteria were defined (see Section 5), during which more issues were identified. First, the main issue and its four identified causes will be explained, after which four additional design requirements are introduced.

The main problem of the current version of the MIRTE Master is the poor performance of the Mecanum wheel locomotion system. Which is caused by the poor system response of the wheel motors. This is an issue as the speed of each wheel is controlled by a PID controller to overcome any irregularities in the environment and the motors. However, these PID controllers cannot be tuned very aggressively due to instability at ‘higher’ gains. These upper gain limits are relatively low and often still result in a barely stable system. The PID gain limitations are a result of the poor gain and phase margin, which indicate the limits of the stable operating region of the system. Four causes for the poor system response have been identified.

First of all, the original wheel velocity signal, as perceived from `ros2_control`, is very noisy. This is caused by the combined effect of the velocity calculation method and the non-synchronized communication between the `ros2_control` hardware interfaces and the `telemetrix` hardware control node. The current velocity calculation method uses the state from the last control loop iteration and the last received encoder state message. The difference between these states is divided by the period of the control loop. This is an issue as the original `ros2_control` frequency is 10 Hz, while the `telemetrix` frequency is configured at 50 Hz. This would be fine in an ideal world, as those frequencies have a common divisor, resulting in exactly 5 messages being received per control loop. However, due to the unsynchronized communication and delays caused by inconsistencies in CPU scheduling and network traffic, this is not always the case, resulting in a larger or smaller radial distance being considered while duration is still considered fixed.

Secondly, the original `ros2_control` hardware interfaces of the MIRTE Master introduced additional discretization to the command signal, which was sent to the `telemetrix` hardware control node. This was done to reduce the CPU usage of the `ros2_control` controller manager by decreasing the number of messages sent. This has been achieved by requiring a minimum absolute motor velocity command of 10 % and by only sending new velocities if any of the new command velocities have changed at least 3 %. While this does reduce the number of messages, it also introduces a variable stepping behavior in the signal. This makes the behavior of the control system depend on the previous velocity commands, making the system more difficult to control as it is no longer time-invariant.

Third of all, the `ros2_control` hardware/controller frequency of the original `ros2_control` setup was configured to run at 10 Hz in order to reduce the CPU usage. This includes the wheel PID controllers and the Mecanum drive controller used for base locomotion. This frequency is relatively low

for this type of control application [2], making fine-grain control difficult. This does not have to be the case, as from a data rate perspective, the controllers could theoretically run at the same rate as the hardware communication node, which currently runs at 50 Hz. However, this is non-trivial to achieve due to non-deterministic communication delays and the absence of loop synchronization. This would also increase the usage of the limited system resources significantly due to the current communication strategy between the `ros2_control` hardware interface and the `telemetrix` hardware control node.

Lastly, the current implementation of the `ros2_control` hardware interface controls the wheel velocity via a client-server model implemented with a ROS 2 service client, which requests the hardware communication node to set new speeds for the motors. This is conceptually simple to follow; however, it has the caveat that a ROS service server³ always sends a response. The current hardware interface doesn't use the response in any way, however this response is still sent by the hardware communication node. This wastes bandwidth and processing time.

3.1 Extra Design Requirements

During the solution exploration phase of the design process, several meetings have been had with the MIRTE development team. In these meetings, possible solution options have been discussed briefly. Based on these discussions, I have identified four additional requirements.

First of all, even though the MIRTE Pioneer and Master share most of their hardware communication software, they currently have independent control setups with their own hardware interfaces. This doesn't initially seem like a strange idea, since they have entirely different locomotion systems. However, the provided hardware interfaces are relatively similar, since they communicate with the hardware in a similar fashion. This duplication of work mainly results in more maintenance effort with an increased risk of bugs.

Second of all, these new `ros2_control` hardware interfaces should be easily configurable, since all robots are unique, as no two motors will ever behave identically, even if they are produced directly after one another. Even if this wasn't the case, versatile robotic platforms, such as the MIRTE Master, could need widely different performance requirements depending on the specific task it is applied to. For example, a robot navigating a busy street might need to react relatively fast with less precision, while a robot stocking shelves needs a much higher positioning accuracy with a less strict response time. This uniqueness in hardware and objectives leads to the need for a generic default configuration, which is easily user configurable.

The current control architecture of the MIRTE robot series does not lend itself well to this, since the configurations are part of the same released binary package as their `ros2_control` hardware interfaces. This results in either having to patch system files, which are overwritten after each minor software update, or requiring to build the package from scratch locally, which is not convenient for users to maintain. Furthermore, some robot-specific constants, like max motor speed, are hardcoded in the compiled `ros2_control` hardware interfaces, making tuning for specific hardware instances even more inconvenient.

Third of all, most of the base locomotion control issues stem from a low controller execution rate and inconsistent communication and execution delays, commonly referred to as jitter. This is partially caused by the MIRTE robot series unique setup, with a separate node to communicate with the hardware. As in more conventional `ros2_control` setups, the `ros2_control` hardware interfaces communicate directly with the hardware instead of introducing an extra layer and communicating via ROS 2's RMW⁴ layer. It was determined that this unique setup should be kept in place to maintain the flexibility of the robots in the MIRTE series.

Furthermore, common mitigation for jitter on general-purpose computers is the use of a real-time or preemptive real-time kernel [12]–[14]. Specific mitigations for ROS 2 setups, including real-time kernels, have been explored in other works [15], [16], however they require a combination of specialized RMW transport layers, hardware that supports Time-Sensitive Networking (TSN) and custom build and tuned real-time kernels. These requirements are not feasible to implement in the general MIRTE Master software distribution, as TSN and real-time kernels require soft- and hardware support for the SBC used in the robot, which is often not available or present on the used budget-oriented models.

³which fulfills the server role in the client-server model

⁴RMW stands for 'ROS 2 middleware (RMW)'

Lastly, it was determined to target the latest long-term support (LTS) ROS 2 version at the time, which was ROS 2 Jazzy. Even though the version supported by the MIRTE team was the previous LTS ROS version, namely ROS 2 Humble. This has been done to future-proof the work and to allow for the utilization of various ROS and `ros2_control` improvements.

4 Proposed `ros2_control` architecture

In this section, the design decisions of the proposed base control system will be explained. These changes have been split up into three categories: `ros2_control` hardware interface, `ros2_control` controller setup, and hardware communication improvements.

4.1 `ros2_control` hardware interface improvements

When using `ros2_control`, it is conventional that the hardware interfaces directly communicate with the hardware. This often leads to a combined hardware interface for the sensors and actuators, a so-called `hardware_interface::System`, because of the complexity introduced by bus sharing over multiple programs.

However, the MIRTE robot series uses a different architecture, where the hardware communication is managed by the `telemetrix` node. This design was a response to the required reconfigurability needed for the MIRTE Pioneer. Furthermore, this also allows for easy hardware control outside of the `ros2_control` framework, which is useful for educational activities. As a result, they opted for `ros2_control` to interact with the hardware via the `telemetrix` node.

In the original `ros_control` for ROS 1 [6], it was a normal practice to combine all robot hardware interfaces into a single module. This resulted in a messy control setup for the original MIRTE Master, which shared a dual-mode hardware interface with the MIRTE Pioneer for the base and a completely isolated control setup for the arm.

This original setup was ported as closely as possible during the initial migration of the MIRTE Master to ROS 2 and `ros2_control`. However, they opted to split the base hardware interface into two distinct hardware interfaces, one for the MIRTE Pioneer controlling two wheels without feedback and one for the MIRTE Master with four wheels with encoder feedback. Furthermore, since `ros2_control`, in contrast with `ros_control`, was designed with loading multiple hardware interfaces for a single robot in mind, there has been opted for sharing a controller manager for the base and arm control setup. A diagram of the original control setup of the MIRTE Master can be found in Fig. 4a.

`ros2_control` is designed with loading multiple hardware interfaces for a single robot in mind, in contrast with `ros_control`. This allows for the creation of configurable robot-agnostic hardware interfaces for (off-the-shelf) sensors and actuators, such as `dynamixel_hardware`[17] for controlling Dynamixel servos and `nmea_hardware_interface`[18] for NMEA-0183 compatible GPS-modules. The main constraint to dividing `ros2_control` hardware interfaces is that communication buses, such as UART and USB serial, cannot be shared across programs or `ros2_control` hardware interfaces.

However, this is not an issue for the MIRTE case as a result of the `telemetrix` hardware communication node. This allows for the (base) hardware interface to split across logical components, each handling one (kind) of hardware device, such as an encoder or a motor. All these new modular hardware interfaces are collected in the new `mirte_modular_hardware` package.

There are some benefits to combining some devices of the same type in one hardware interface, such as a reduced node endpoint count and the ability to manage subsystems easier. Another reason is a merged interface for synchronization purposes; one such example is the proposed `MultiMotorActuator`, which sends a single message for multiple motors to ensure synchronization, which is essential for achieving a reliable Mecanum drive system [9]. For the recommended proposed encoder interface `EncoderSensor`, it was chosen to not support multiple encoders per interface to simplify the implementation of this interface, since the benefits are limited. A full diagram of the proposed control architecture can be found in Fig. 4b.

Another improvement that has been made to the proposed system is the usage of ROS topics instead of ROS services. While using service requests to set the motor speed is conceptually simpler than the proposed alternative because of their simplicity and the guarantee the request will be received

by someone⁵. However, this comes with the added cost of having to handle the response in the RMW layer even if the response is ignored by the node itself, which limits the available throughput and adds an undesired and unnecessary communication load.

The alternative of using topics instead has multiple benefits over the use of services, as topics allow for unconditional sending of the messages regardless of if somebody is listening or the message arrives. This has the benefits of introducing less system load at the same frequency and the ability to configure it to not retry sending the message, which is unnecessary at higher frequencies. This has the slight downside of it being possible that nobody is acting on the speed request message, which could be checked for manually if desired.

Furthermore, improvements were made to the wheel velocity calculations. In the original hardware interface the wheel velocity was calculated, based on the difference of ticks between the latest message and the last control loop iteration, which was divided by the loop time. This made the velocity calculation dependent on the frequencies of hardware interfaces and `telemetrix`, since if these didn't divide exactly the velocity would have a lot of noise. For example, in Fig. 8c, where `ros2_control` was running at 20 Hz and `telemetrix` at 50 Hz, this results in an average of 2.5 messages per control loop. In practice, this meant that every other control loop had a newer message compared to the last control loop (the 3rd message instead of the 2nd), since it was adjusting for this inconsistency every other velocity measurement would be higher than the measurement before, causing an oscillation. To mitigate this in the proposed hardware interface, the velocity was calculated on the difference between the last two received encoder messages, which was divided by their time difference. This results in a frequency-independent time measurement with less noise (as can be seen in Fig. 8d).

Also, more configurability has been introduced on a per-wheel basis, allowing for different wheels to have different max velocities and encoder tick counts per revolution.

4.2 `ros2_control` controller setup improvements

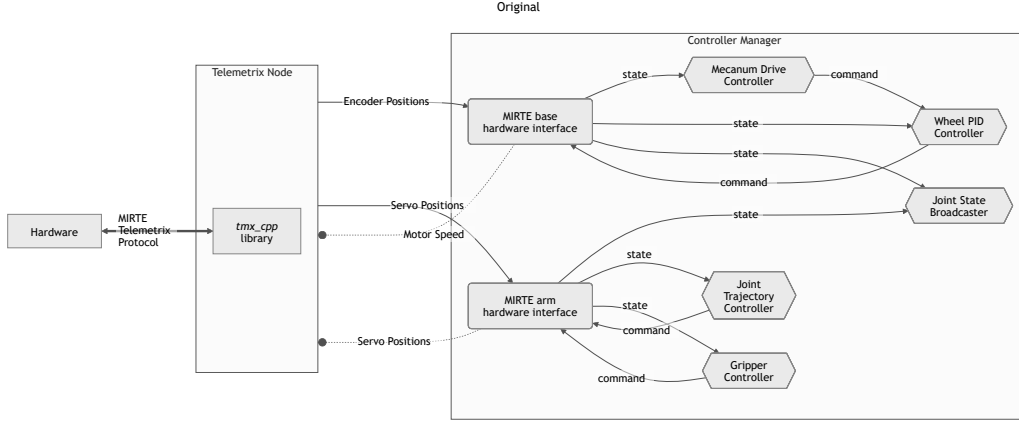
The `ros2_control` setup of the MIRTE Master also includes several controllers, of which the proposed changes will also be discussed here. `ros2_control` supports controller chains, which allows for a motor to be speed controlled via a PID controller between the hardware interface and the overall locomotion controller. In this section, the suggested controller setup changes will be discussed.

On ROS 2 Humble, the MIRTE Master used to ship with the `MecanumDriveController` from the `clearpath_mecanum_drive_controller` package[19]; however, this controller is not available for ROS 2 Jazzy. This controller has been replaced by a nearly identical version maintained by the `ros2_control` team (`mecanum_drive_controller/MecanumDriveController`) [20]. Using this version has the benefit of guaranteed compatibility with future versions of `ros2_control`.

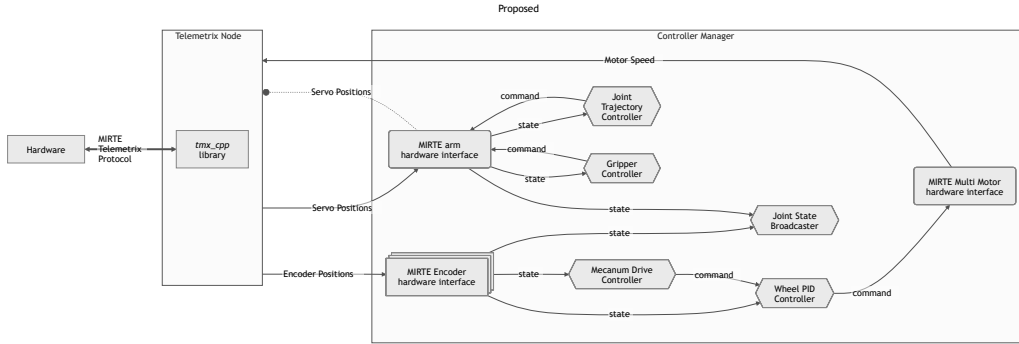
At the beginning of this thesis, the current version of the MIRTE Master shipped with an 'untuned' PID controller with all gains set to 0 except for the proportional gain, which was set to 1. This essentially disables the PID controller, with the intention that it should be tuned by the user of the robot. Leaving the exact tuning of the wheel PID controllers to the end user is a really defensible decision, as the exact requirements can differ per hardware instance and objective. However, the PID configuration was shipped as part of the binary package in the original setup, making it difficult to tune permanently during deployment, requiring the parameters to be loaded manually at every run or patching the installed file owned by root, with the caveat that it will be overwritten when the package gets updated. Therefore, I have separated the hardware interfaces from the controller configurations in the proposed `ros2_control` setup, which still allows the hardware interfaces to be shipped as a binary package while leaving the configuration-only control setup package to be built directly on the robot, allowing for simpler controller configuration changes.

Lastly, in the original control setup, the controllers were configured to run at 10 Hz, which is relatively slow for locomotion interfaces. The theoretical maximum control rate is equivalent to the `telemetrix` hardware communication rate, which is currently configured at 50 Hz. For the proposed solution, I attempted to increase the frequency towards this theoretical maximum, as it would allow signals with frequency components up to 25 Hz to be reconstructed completely according to the Nyquist-Shannon sampling theorem[3].

⁵This is because a service-client request is a one-to-one communication channel, which requires the server to participate and respond.



(a) The original `ros2_control` architecture of the MIRTE Master, which uses a ROS service to control the speed of the base motors, allows for synchronization as a result of the single request while inducing extra load because of the required, but unused, service response.



(b) The proposed `ros2_control` architecture of the MIRTE Master, which uses a single ROS topic to control the speed of the base motors, allows for the same synchronization as the original implementation while being more computationally efficient, as no response is send.

Figure 4: `ros2_control` setup diagrams, where the white boxes are ROS 2 nodes, the rounded grey boxes are `ros2_control` hardware interfaces, the stacked grey boxes are multiple `ros2_control` hardware interfaces of the same type, and the grey hexagons are `ros2_control` controllers. The connections mean the following: the thick double-headed arrow is a UART connection, the normal arrows, which are connected to nodes, are ROS topics (the arrowhead points to the subscriber), the dotted arrows with round ends are ROS services (pointing towards the service server), and the normal arrows within the Controller Manager are `ros2_control` command/state interfaces.

However, while attempting and testing this with the measurement methods that will be discussed in Section 5, I ran into an issue where the encoder state messages were not arriving on schedule. The jitter in the ROS message stream is caused by the nondeterministic nature of network traffic, which is used by most of the RMW implementations under the hood. This limited the maximum feasible control frequency to approximately 20 Hz. The hardware interface and controllers for the arm have been capped to the original 10 Hz to reduce system load, since the control setup for the arm is outside of the scope for this thesis and therefore has remained mostly untouched.

4.3 Hardware communication improvements

Four optimizations and tweaks to the `telemetrix` hardware communication layer were also necessary to make some of the before-mentioned proposals feasible and/or more effective. This will be discussed in this section in no particular order.

Firstly, in order to make the recordings of the measurement as complete as possible, ROS' service introspection [21] needed to be used. This feature was introduced in ROS 2 Iron and allows introspection of the one-to-one service communication for debugging and recording purposes. When service introspection is enabled, the service calls and responses can be recorded and replayed using `rosviz2`. For this work, a parameter toggle per `telemetrix` device was introduced to enable service introspection for the measurement trials, which used those services.

Secondly, in order to optimize the message traffic of the encoders, an update-only topic was introduced. This topic only sends a message when new encoder data from the microcontroller has been received, where normally a timer is used in addition to this to resend older data. This limits the message traffic, as only the changes were necessary to calculate the wheel velocity.

Third of all, an adaptation to `telemetrix` was necessary in order for the change from ROS services to ROS topics for motor control to be possible. Individual motors already had a topic interface to control their speed; however, this was insufficient for the Mecanum drive locomotion system of the MIRTE Master. This is the case because a Mecanum drive system's omnidirectional movement is controlled by the wheel speed relations, making command synchronization essential for consistent and reliable movement. Therefore, it is beneficial to send a single message with all the motor commands, which prevents such synchronization issues. This also comes with the added advantage of having to do the somewhat costly serialization once instead of four times, which helps to reduce the overall system load.

Finally, command synchronizing based on a single message and service call in the current iteration of the MIRTE Master is straightforward as the motors are all connected to a PCA9685 multi-channel PWM driver. The PCA9685 is connected via an I2C interface, which exposes a way to set the duty cycle of multiple sequential pins simultaneously. However, the new hardware revision of the MIRTE Master, which is currently in one of the final stages of development, removes the PCA9685 module and will connect the motor drivers directly to the microcontroller. This could reintroduce the before-mentioned command synchronization issues, as controlling multiple motors with a single message or service call was only supported for motors connected to an external module in the original implementation. To keep the results of this work relevant, I opted to implement a 'virtual' `telemetrix` hardware device, which added this functionality for motors, which have been added directly instead of via a module.

5 Designing a measurement method

The system characteristics needed to be measured in order to verify the proposed improvements and identify the behavior changes. However, there is no standardized `ros2_control` tooling for measuring system responses, such as the frequency or step response. Therefore, a custom solution needed to be created, which is light-weight as to influence the data as little as possible. In this section, the used measurement setups will be discussed. First, the general structure of the measurement setup will be explained, after which the measurement type-specific tweaks will be discussed.

5.1 General

Because the focus of this thesis is on the effect of the proposed hardware interface changes, the measurements should measure the hardware interface's command and state interfaces⁶ as directly as possible.

⁶Command and state interface as the `ros2_control` in- and outputs for the system

A measurement should also collect as much relevant system data as possible, in order to be able to reason about the main measurement data. In our case, the command interfaces are the desired wheel velocities, while the state interfaces are the measured wheel angle and velocity. The other parts of the hardware chain we are interested in are the commands the hardware interface sends to `telemetry`, which intern will command the hardware, and the encoder state readings used in the states, the positions and velocities. A custom `ros2_control` controller needs to be used to drive the measurement in order to ensure the measurement only includes the `ros2_control` hardware interfaces, the `telemetry` hardware communication and the hardware behavior itself.

Before using the encoder readings of the system, it is essential to ensure these measurements never miss an encoder tick. Therefore, an analytical verification has been done to verify the consistency of these readings, which can be found in Appendix A.

All the involved data can be recorded using the `rosbag2` tooling [22], which can be used to record ROS communication. Most of the `telemetry` provided data was already exposed via topics, making it simple to integrate in the recording. The `ros2_control` command and state interfaces are also exposed via the `controller_manager/introspection_data/**` topics, which provides all the `ros2_control` interface names and values in an efficient format. `rosbag2` can even record services if service introspection has been setup, which has been done for the `telemetry` services being used. The following relevant topics have also been recorded for every test:

/rosout The default ROS logging topic, where all log messages are sent.
/diagnostics This standardized ROS diagnostic messages, where all diagnostics messages are sent[23].
controller_manager/activity A topic, containing the current `ros2_control` hardware interface and controller state, only updates when a state transitions occur.
<controller>/transition_event A topic containing the ROS lifecycle transition event messages of the controller.

Most measurements consist of multiple trials in order to see the effect of configuration changes or to eliminate some variability of the system. Therefore, it makes sense to partially automate the measurement procedure, so all trials can be measured consistently concerning timing and setup. This process can only be partially automated, since changing `ros2_control` hardware interface configuration requires a restart of several subsystems. Fully automating this system would require changing the system setup, which would affect the measurements, and therefore it was excluded as an option. This measurement director oversees setting up the recording, and configuring and activating the custom controller for the measurement. This ensures that the data is recorded accurately, labelled correctly and stored with sufficient metadata to reconstruct the system configuration.

In order to support the easy measurement configuration, the ROS Parameter iterator system was created. This system allows parameters to be iterated dynamically with different iteration methods. These individual parameter iterators can be combined by a product iterator, which will iterate over all possible combinations of the iterators. Currently, the following types of parameter iterators have been implemented:

Fixed An ‘iterator’ over a single fixed value.
List An iterator over a provided parameter array.
Linear An iterator from a starting value to an end value in linear steps with a given step count.
Log An iterator from a starting value ($\text{base}^{\text{start}}$) to an end value (base^{end}) in logarithmic steps with a given step count.

Another important task of the measurement director⁷ is ensuring the recording is started and stopped at the correct time. The beginning of the recording should capture enough data so that the data could be reindexed in a way where the controller activation happens at 0s. Having a correct time relative to the controller’s activation is crucial since the tests have been set up in a way where the controller’s time reference is reset on activation. Having an accurate reference for this is necessary in order to measure accurate time delays, which are a crucial part of determining the system response.

Initially, this was planned to be done using the lifecycle transition event message timestamp, which gets published as a side effect of the controller activation. However, as described in the GitHub issues `ros2/rcl#1019` [24] and `ros2/rclcpp#564` [25], the timestamp field is not being filled. This limits the

⁷The `response_director` node in the `wheel_response_recorder` package contained in the repository of corresponding with this thesis as described in Appendix C.

use of such a transition event message for synchronization use cases, as the other available time fields (send and receive time) are not precise enough for this purpose. This problem has been resolved as an indirect result of the work for ROS 2 Rolling as described in Appendix C.1. However, it was not feasible to use the patch for this thesis due to how far down in the dependency tree the involved libraries are.

To resolve this, measurement-specific synchronization strategies have been used, which use recognizable events in the input signal to reindex the data if available. A universal strategy, which has been applied as an initial offset correction on all measurement types, is to use the timestamp of the controller activation log message. This message is published internally, reasonably close to the activation callback, and therefore already gives an acceptable result.

5.2 Sinusoid-based Frequency Response

There are multiple ways to determine the frequency response of a system, a fairly straightforward way is to measure the system response of a sinusoid at different frequencies. From this measurement data, the phase shift and magnitude of the output with respect to the input can be found by fitting a sinusoid with a free gain (factor) and phase offset, while using the same frequency and offsets as the input. Such a fit should be able to capture the output exactly, when a single frequency input is being used.

A well-known limitation of the MIRTE Master is the significant motor speed dead zone when starting from standstill, this is a result of the motor's internal friction. To identify the effects of this dead zone in the measurements, two measurement types were conducted.

The first measurement used a sinusoid with a magnitude larger than the dead zone, this is the most standard way of measuring such a sinusoid. This method still captures the full system behaviour from standstill. However, it fails to capture local effects on low speeds.

The second measurement used a smaller magnitude, but with an offset to ensure the input is always above the dead zone. This method eliminates the effects of the dead zone. However, full system reversal is not observed. Only the dynamic motor friction plays an effect in this measurement, as the motor does not 'stop' spinning, as a result of the offset ensuring that the minima of the sinusoid are above the dead zone.

Such an offset sinusoid has an additional frequency component of 0 Hz and a phase delay of 0.5π rad, which needs to be compensated for. This data has also been obtained experimentally by measuring a constant system to get this static gain at 0 Hz.

To ensure a good fit of the sinusoid can be made, the measurement time was set to at least 5 s or 2 full periods. The period limit was put in place to ensure at least enough of the signal was captured to make an unambiguous reconstruction. However on shorter periods, recording time inaccuracies will start to interfere with the measurement, therefore a minimum recording duration of 5 s was put in place.

Ideally, the measurements should be taken with the robot driving on the floor because then the dynamics of the whole robot are taken into consideration. However, this is nontrivial with these two measurements, as they drive for a relatively long time. In the zero-offset case, the robot would theoretically move only back and forth, however due to minor physical motor differences and floor inconsistencies, the robot ends up slowly spinning and thereby increasing the required floor space. In the offset case, the robot drives forwards continuously, which also requires a large space, however since the robot is unregulated, it does not compensate for its inaccuracies, and therefore tends to drift to a single side, increasing the required space even more. Since the required spaces for these measurement types needs to be unobstructed for an accurate system response, the area necessary is not realistic to acquire. Therefore, I have chosen to only do these measurement kinds when the robot is 'lifted', meaning that the wheels did not contact the floor during the measurement.

5.3 Step Response

In order to verify the previous measurement, another measurement type needed to be conducted. Measuring the step response of a system can give a lot of insight into the system response, since it can be related back to the impulse response. Such a measurement has a step function as input, instantly rises the input to a certain level and remains constant after that moment. A step response is also a relatively fast measurement to do, allowing for multiple trials of identical configurations, so outlier runs can be detected.

This measurement was done at several input wheel velocities, repeating each velocity 5 times alternating it with its negated value. This has been done in order to equalize the effects of encoder/motor play in the measurements and to allow for convenient measurements on the floor (the ‘grounded’ configuration), since the robot will move back towards the starting position of the next measurement. The measurement duration was determined to be 7 s when the robot is lifted and 6 s when the robot is grounded. The step event occurs at 2 s for both setups. This allows for recording the background system data before the control system is commanded to move to, which can be used to compare system resource usage, such as CPU and RAM utilization, between the idle and busy state of the robot in the different control configurations. The remaining duration after the step event captures the full transient system response in both cases, after which the system settles at approximately constant wheel speeds. This measurement has been conducted in a ‘lifted’ and ‘grounded’ configuration because it has a relatively short and fixed duration, allowing the additional ‘grounded’ measurements due to the limited space requirement.

The ‘lifted’ configuration can be used to validate the lifted results of the sinusoid-based frequency response measurements due to their similar setup. Whereas, the ‘grounded’ result can be used in combination with the ‘lifted’ results to relate the ‘lifted’ scenario with the more realistic ‘grounded’ scenario.

An average behavior can be created by averaging identically setup runs, which are available because of the 5 identically configured trials. However, such averaging requires exact time indexing with respect to the step event. Otherwise, the reference will not be a true step function, as the averaged version of it includes a non-instant rise time.

To exactly synchronize the time indexes of the step response measurement, the rising edge of the step event has been used. This is an ideal synchronization point, since even though the recording did not start directly with the controller, that event is still at a fixed ‘time point’ in the measurement. This allows for a correct sync of the system responses, in the case that it can be ensured that the first time step where the step has occurred has been recorded. This is something which needs to be checked because a handful of introspection data messages over all tests have been lost. It has manually been verified that no introspection data messages around the step event time have been lost.

6 Measurement Results

The results from the measurements described in Section 5 will be analyzed in this section. All measurements have been done in the following four distinct hardware interface configurations:

- The original hardware setup where all controllers run at 10 Hz.
- The original hardware interfaces, with the controllers and hardware interfaces related to the Mecanum wheel drive running at 20 Hz, while the arm-related hardware interfaces and controllers are limited to 10 Hz.
- The proposed hardware setup with all controllers and hardware interfaces running at 10 Hz.
- The proposed base hardware interfaces, with the controllers and hardware interfaces related to the Mecanum wheel drive running at 20 Hz, while the arm-related hardware interfaces and controllers are limited to 10 Hz.

The amount of data and figures generated from these measurements is considerable, as most measurements are done on a per-wheel basis. A comparison between the individual wheels was deemed irrelevant, as the exact results differ per motor and hardware instance. While these results differ, as they are dependent on the exact motor characteristics of each individual motor, all measurements show similar behavior per configuration. Therefore, there has been opted to only discuss the analysis of the front left wheel in this section. Additionally, all figures of the configurations running at 10 Hz have been placed in Appendix B in order to facilitate clarity. The omitted figures have been made available online as described in Appendix C.

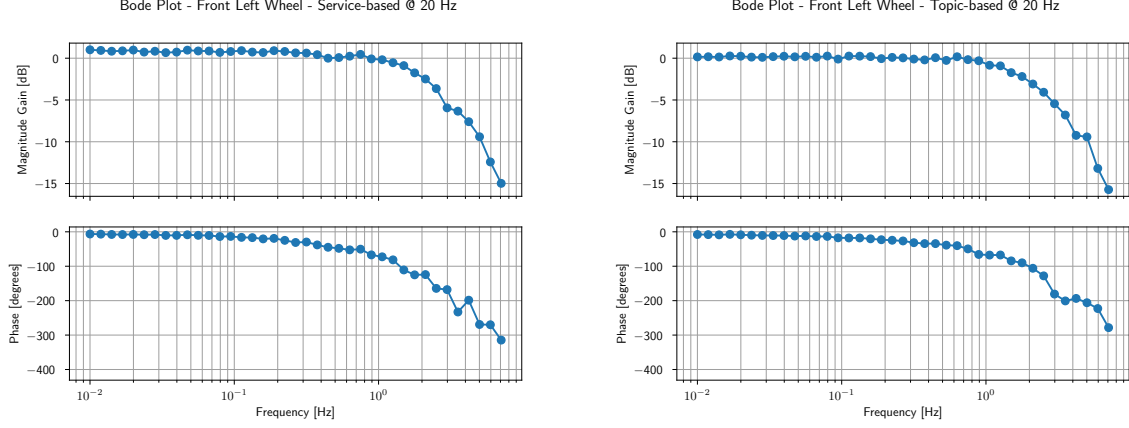
6.1 Frequency response

In this section, the measurement results of the frequency response will be discussed. Each measurement consists of multiple trials, which drives the setup with sinusoid using a different frequency for each trial, in order to capture the usable frequency spectrum of the control setup. This measurement has two

variations, with and without an offset. This has been done to identify the effect of the motor dead zone, which is a result of internal motor friction.

6.1.1 No Offset

The bode plots resulting from this test of the front left wheel can be found in Figs. 12 and 5 for 10 and 20 Hz respectively.



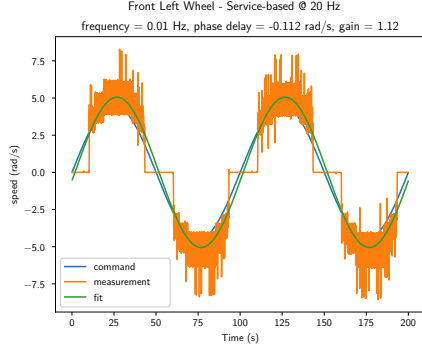
(a) Frequency response of the original service-based hardware interfaces at 20 Hz using a sinusoid around zero. From this graph, it can be found that the gain and phase margin are 6 dB and 110° in this configuration.

(b) Frequency response of the proposed topic-based hardware interfaces at 20 Hz using a sinusoid around zero. From this graph, it can be found that the gain and phase margin are 6 dB and 135° in this configuration.

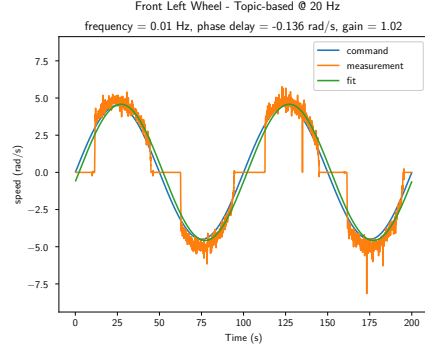
Figure 5: Frequency response of the hardware interfaces generated using a sinusoid around zero while being lifted with a `ros2_control` controller frequency of 20 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 20 Hz, the maximum measurable frequency is limited to below 10 Hz as described in the Nyquist-Shanon theorem [3]. In the subfigures it can be seen that the proposed topic-based `ros2_control` setup has a larger phase margin compared to the original service-based setup.

The magnitude-bode plots are mostly consistent between the original and proposed setups, especially at higher controller frequencies. The original setup at 10 Hz (Fig. 12a) has a significantly lower gain margin of 2.6 dB when compared to the other trials. The other trials have a gain margin of approximately 6 dB (see Figs. 5a, 5b and 12b). Demonstrates an interesting effect as increasing the `ros2_control` hardware frequency or using the proposed `ros2_control` hardware setup (at any of the tested frequencies) has a similar result on this aspect. This higher gain margin in these different setups is a result of a ‘smaller’ time delay in the wheel velocity signal. In the original implementation, the velocity was calculated based on the wheel position from the previous control loop iteration and the last received message, with the assumption being made that the last message is dated a single period after the last calculation. That assumption does not hold at lower frequencies, such as 10 Hz, since there are no guarantees about when this last message was received, resulting in a velocity that can lag, after which it will catch up (as the ‘unused’ encoder ticks will be used for the next time step). The proposed hardware interface is not effected by the `ros2_control` frequency as it uses the two last received encoder (update) messages for the velocity calculation, in which their position difference gets divided by the time difference from the measure header, resulting in something closer to the instantaneous wheel speed.

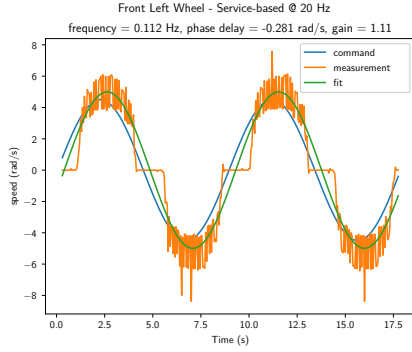
From the phase-bode plots, a similar situation can be seen. Where the original setup at 10 Hz has the smallest phase margin of 70°. When the velocity calculation is improved by a single method, either by shorter timesteps (a higher `ros2_control` frequency) or by using the proposed method, the phase margin increases to approximately 100°. When the proposed method is used at a controller rate of 20 Hz, the phase margin increases even further to 135°. This suggests that the ‘effective’ delay decreases as a result of the frequency increase and the proposed setup improvements are equally effective individually,



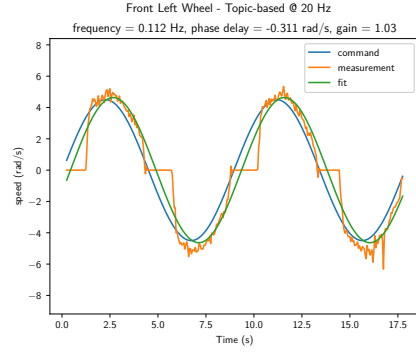
(a) Original service-based **ros2_control** hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 0.01 Hz



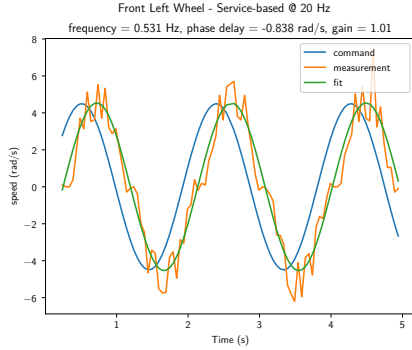
(b) Proposed topic-based **ros2_control** hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 0.01 Hz



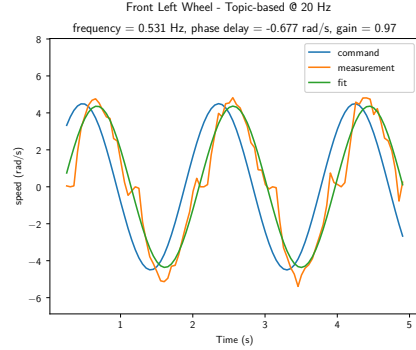
(c) Original service-based **ros2_control** hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 0.112 Hz



(d) Proposed topic-based **ros2_control** hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 0.112 Hz

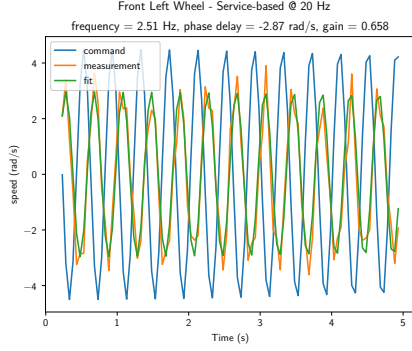


(e) Original service-based **ros2_control** hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 0.531 Hz

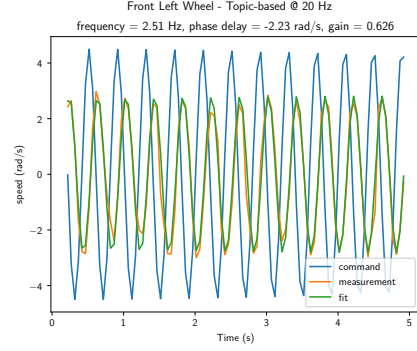


(f) Proposed topic-based **ros2_control** hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 0.531 Hz

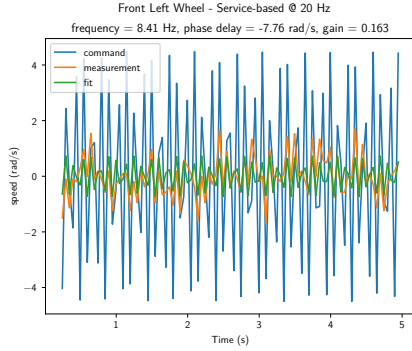
Figure 6: The individual trial fits for the frequency response measurements generated using a sinusoid around zero while being lifted with a **ros2_control** controller frequency of 20 Hz for the Mecanum drive-related components. Due to the **ros2_control** frequency being 20 Hz, the maximum measurable frequency is limited to below 10 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency and amplitude as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 5. In these subfigures it can be seen that the proposed topic-based setup has a finer command discretization and a smaller command deadzone compared to the original service-based setup.



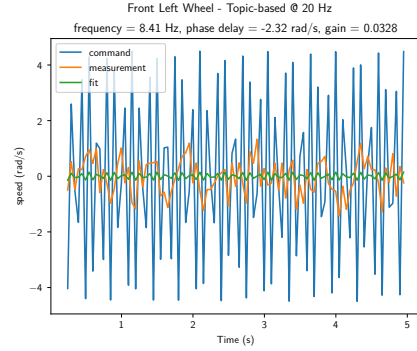
(g) Original service-based `ros2_control` hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 2.51 Hz



(h) Proposed topic-based `ros2_control` hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 2.51 Hz



(i) Original service-based `ros2_control` hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 8.41 Hz



(j) Proposed topic-based `ros2_control` hardware interfaces running at 20 Hz, driven with a sinusoid with a frequency of 8.41 Hz

Figure 6: The individual trial fits for the frequency response measurements generated using a sinusoid around zero while being lifted with a `ros2_control` controller frequency of 20 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 20 Hz, the maximum measurable frequency is limited to below 10 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency and amplitude as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 5. In these subfigures it can be seen that the proposed topic-based setup has a finer command discretization and a smaller command deadzone compared to the original service-based setup. (cont.).

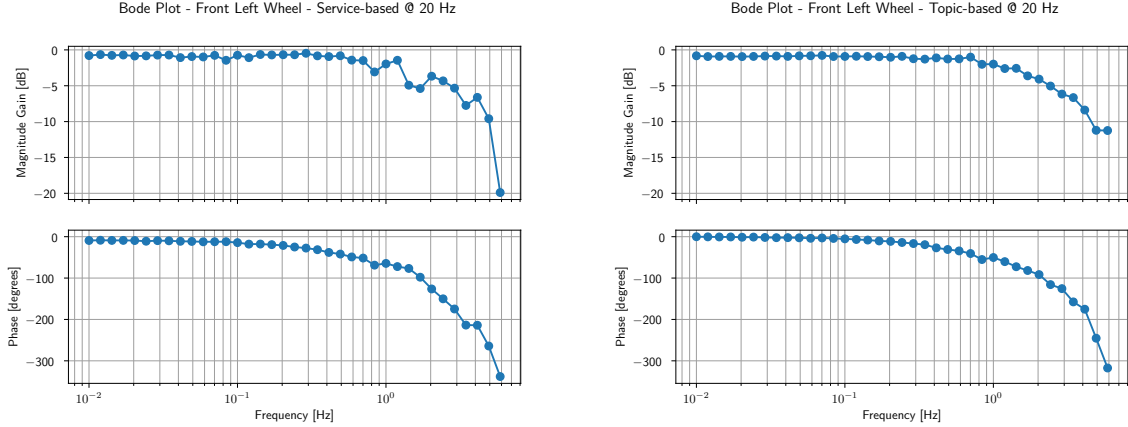
while also allowing for the combination of both, which results in a larger improvement. This last fact shows that both methods are equally valid.

The effects of the wheel velocity calculation improvements can also be seen in the individual signal measurements. The original hardware interface had implemented a minimum velocity change to reduce the CPU load, which in it turn caused significant aliasing, as can be seen in Figs. 6a and 13a. This aliasing results in quantized velocity plateaus that are coarser than the `telemetrix` hardware communication support. These large distinct levels are not present when using the proposed hardware interfaces (see Figs. 6b and 13b), since it will only send a message when the requested wheel speeds results in a different `telemetrix` motor command. All configurations do show an initial motor speed deadzone, below which the motor cannot overcome the internal friction (See Figs. 6a, 6b, 13a and 13b). In all tests, some deadzone will be present, as this is a result of the physical motor properties. However, the original `ros2_control` setup had a fixed built-in minimum threshold to prevent the motor from burning out (see Figs. 6a, 6c, 13a and 13c). This lower limit was often too strict as it was based on moving the robot from standstill, but in the case where the wheels are already turning, the lower hardware speed limit decreases, allowing for slower speeds. The proposed `ros2_control` hardware interfaces don't have such a lower protection limit, allowing the wheels to spin at lower speeds when provided when they are already spinning (see Figs. 6b, 6d, 13b and 13d).

Furthermore, the noise has significantly been reduced in the proposed `ros2_control` setup due to the wheel velocity calculation improvements mentioned earlier, as can be seen in Figs. 6 and 13.

6.1.2 Offset

The bode plots resulting from this test of the front left wheel can be found in Figs. 14 and 7 for 10 and 20 Hz respectively.

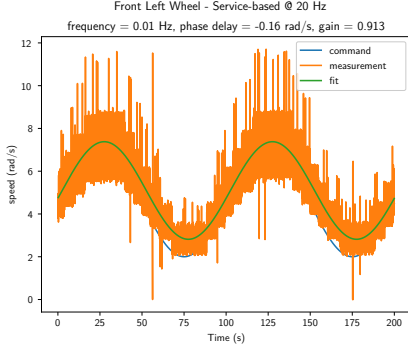


(a) Frequency response of the original service-based hardware interfaces at 20 Hz using an offset sinusoid. From this graph, it can be found that the gain margin in this configuration is 5.1 dB. The phase margin cannot be determined as the graph is always below 0 dB in the measured range.

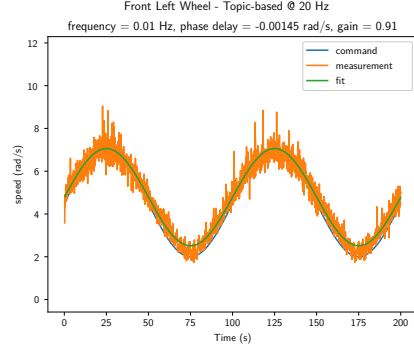
(b) Frequency response of the proposed topic-based hardware interfaces at 20 Hz using an offset sinusoid. From this graph, it can be found that the gain margin in this configuration is 8.2 dB. The phase margin cannot be determined as the graph is always below 0 dB in the measured range.

Figure 7: Frequency response of the hardware interfaces generated using a sinusoid with an offset of 4.5 rad s^{-1} while being lifted with a `ros2_control` controller frequency of 20 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 20 Hz, the maximum measurable frequency is limited to below 10 Hz as described in the Nyquist-Shanon theorem [3]. In the subfigures it can be seen that the proposed topic-based `ros2_control` setup has a smoother bode magnitude plot with a larger gain margin compared to the original service-based setup.

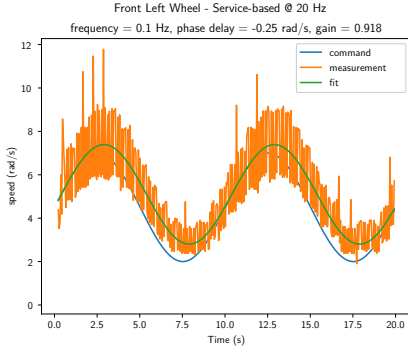
The magnitude-bode plots are not as smooth as in the previous offsetless test (see Section 6.1.1), however they do hold a consistent shape between one-another where most artifacts are present in every configuration, all be it at a lesser severity. An example of such an artifact is a bump in the magnitude somewhere between 0.1 Hz to 1.0 Hz, which can be seen in all bode plots of this test (See Figs. 7 and 14).



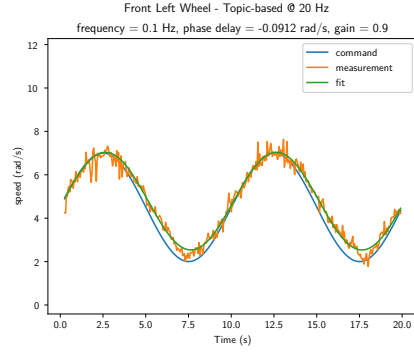
(a) Original service-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 0.01 Hz



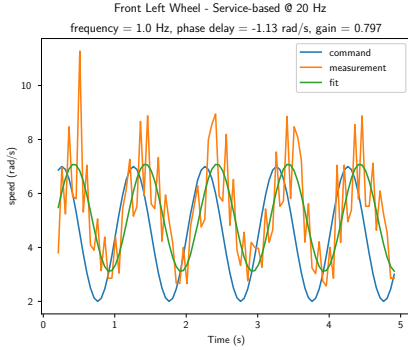
(b) Proposed topic-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 0.01 Hz



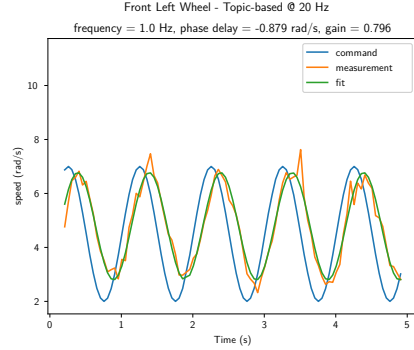
(c) Original service-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 0.1 Hz



(d) Proposed topic-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 0.1 Hz

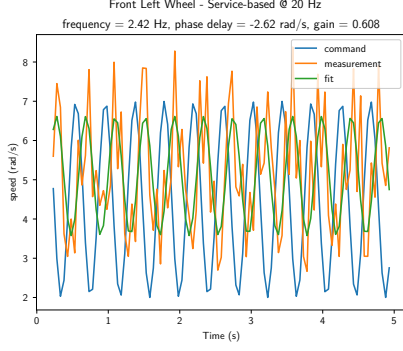


(e) Original service-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 1.0 Hz

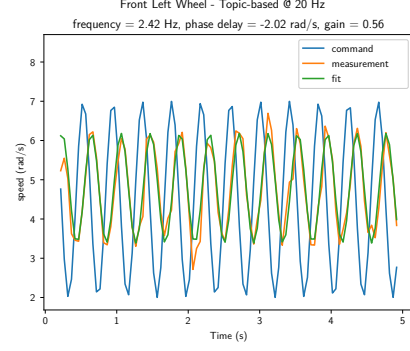


(f) Proposed topic-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 1.0 Hz

Figure 8: The individual trial fits for the frequency response measurements generated using a sinusoid with an offset of 4.5 rad s^{-1} while being lifted with a `ros2_control` controller frequency of 20 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 20 Hz, the maximum measurable frequency is limited to below 10 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency, amplitude and offset as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 7. In these subfigures it can be seen that the proposed topic-based setup has significantly less noise in the velocity state signal compared to the original service-based setup.



(g) Original service-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 2.42 Hz



(h) Proposed topic-based `ros2_control` hardware interfaces running at 20 Hz, driven with an offset sinusoid with a frequency of 2.42 Hz

Figure 8: The individual trial fits for the frequency response measurements generated using a sinusoid with an offset of 4.5 rad s^{-1} while being lifted with a `ros2_control` controller frequency of 20 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 20 Hz, the maximum measurable frequency is limited to below 10 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency, amplitude and offset as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 7. In these subfigures it can be seen that the proposed topic-based setup has significantly less noise in the velocity state signal compared to the original service-based setup. (cont.).

Furthermore, the gain margins for this offset configuration tell a very similar story as the offsetless tests as described in Section 6.1.1. Where the original setup running at frequency of 10 Hz (Fig. 14a) has the lowest gain margin of 3.5 dB when compared to the other configurations. If a single improvement is implemented, so either raising the `ros2_control` frequency to 20 Hz (Fig. 7a) or using the proposed `ros2_control` hardware interfaces (Fig. 14b), the gain margin improves to around 5.1 dB. The best gain margin of 8.2 dB can be observed when both suggested improvements are implemented, which means the system is using the proposed `ros2_control` hardware interfaces at a frequency of 20 Hz (Fig. 7b). This again means that both suggested changes, increasing the `ros2_control` frequency and using the proposed hardware interfaces, are individually beneficial in improving the gain margin, and thereby its stability because an increased gain margin can be beneficial to achieve stability at more aggressive controller gains.

The phase margin for this test configuration falls outside of the measured frequency range and will therefore not be discussed in great detail. However, it is interesting to note that without the effects of the motor speed deadzone caused by its internal friction, the phase margin increases. This is more likely to be the result of inaccuracies in the speed mappings between `ros2_control`, `telemetrix` and the actual hardware.

When inspecting the individual sinusoid fits (Figs. 8 and 15) used for extracting the data required to make the bode plots it might be suggested that the original setup has a slightly higher gain at lower frequencies. However there are two other findings which make this original setup less favourable. First of all, the phase delay is often significantly higher in the cases where the gain is higher when using the original instead of the proposed hardware interfaces, as can be seen when comparing Figs. 15a and 15b, Figs. 15c and 15d, Figs. 8a and 8b or Figs. 8c and 8d. Second of all, the original `ros2_control` hardware interface has a very noisy wheel velocity reading, as explained in Section 4.1. This can be especially well observed in Figs. 8a, 8c, 15a and 15c. The peaks appear to be a lot higher when using the original hardware interfaces as a result of this noise; however, in reality, these peaks are unobservable in the output as they are calculation artifacts. The combination of this inflated wheel speed measurement and the higher phase delay makes the original hardware interfaces unfavorable to use.

6.2 Step response

In this section, the measurement results of the step response will be discussed. Each measurement consists of multiple trials, which drives the setup with a scaled setup function with specific scaling for each trial. Each step size has been tested 5 times alternating with its negated counter part in order to make floor measurements feasible as the robot would roughly return to its starting positions and equalize the effects of play in the motor gearboxes and encoders. This also enables the possibility to analyze the average trial created from all measurements with the same step size. The step sizes (and their negative counterparts) that have been tested are 3.0 rad s^{-1} , 4.5 rad s^{-1} and 21.0 rad s^{-1} .

These values have been chosen because of the following reasons: 3.0 rad s^{-1} is slightly above the deadzone speed minimum of the original hardware interfaces such that all wheels can start turning from standstill when the used MIRTE Master was lifted⁸; 4.5 rad s^{-1} is the amplitude of the frequency response measurements with no offset (as described in Section 6.1.1); and 21.0 rad s^{-1} as it is the theoretical maximum wheel speed and is therefore being mapped to the maximum `telemetrix` motor speed value.

This test has two variations: lifted and on the floor, or ‘grounded’.

The lifted configuration has been made to validate the results of the frequency response measurements (Section 6.1). The properties from both tests should be very close, as both measurements are performed with the wheel wheels lifted off the ground.

The grounded variation has been measured in order to attempt to relate the other ‘lifted’ measurements to the actual driving situation. This can be done by attempting to find similarities between the step responses. Furthermore system resource usage data have been recorded for this variant in addition to the standard collected data.

6.2.1 Lifted

In this section, the result of the lifted step response measurement will be discussed. The step response graphs resulting from this test of the front left wheel can be found in Figs. 16 and 9 for 10 and 20 Hz respectively.

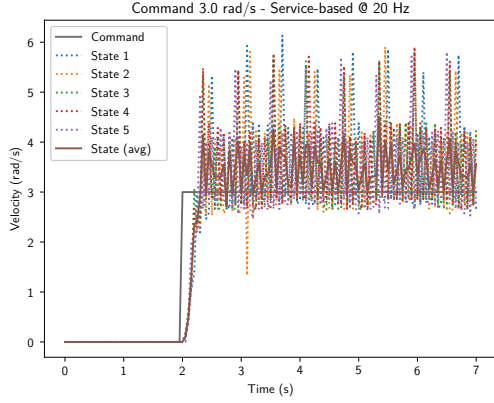
Table 1: The final wheel velocity averages of the lifted front left wheel recordings. These averages are taken over the interval starting 1 s after the step event until the end of the recording.

Configuration	Commanded Step Size		
	3.0 rad s^{-1}	4.5 rad s^{-1}	21.0 rad s^{-1}
Service Based @ 10 Hz (Original)	3.60 rad s^{-1}	4.98 rad s^{-1}	18.2 rad s^{-1}
Topic Based @ 10 Hz	3.41 rad s^{-1}	4.71 rad s^{-1}	18.3 rad s^{-1}
Service Based @ 20 Hz	3.48 rad s^{-1}	4.89 rad s^{-1}	17.2 rad s^{-1}
Topic Based @ 20 Hz (Proposed)	3.51 rad s^{-1}	4.73 rad s^{-1}	18.2 rad s^{-1}
Average	3.50 rad s^{-1}	4.83 rad s^{-1}	18.0 rad s^{-1}

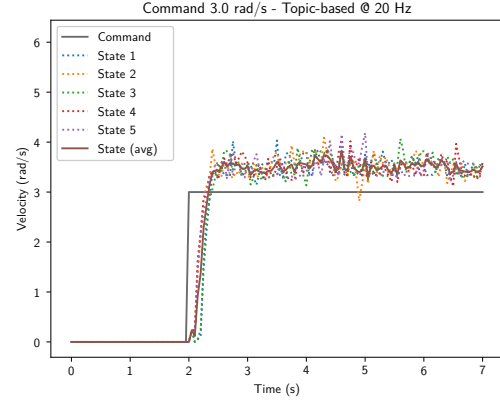
It can be seen that average final wheel velocity per step size is a similar value across all configurations, as can be seen in Table 1. If these values would most likely be closer together if the average were taken over a larger interval. However, since for each step size the values roughly correspond, it shows that the physical hardware response dominates the steady-state response of the wheel control system. At the same time, the individual differences between all configurations show that the state measurements are influenced by noise. In Table 1, it can also be seen that the introduction of the control frequency-independent velocity calculation in the topic-based configurations results in these averages being closer at the different control frequencies compared to the original service-based configurations.

This is the result of the noise reduction, as can also be seen in Figs. 9 and 16. It can be seen in Figs. 9a, 9b, 16a and 16b that the final velocities of all trials with a step size of 3.0 rad s^{-1} are close to 3.50 rad s^{-1} , while the topic-based configurations (Figs. 9b and 16b) experience less noise in the signal. A similar effect can be seen for trials at 4.5 rad s^{-1} and 21.0 rad s^{-1} (see Figs. 9c, 9d, 16c

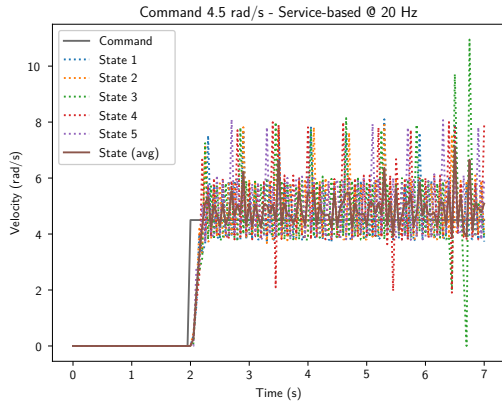
⁸This can differ depending on the specific MIRTE Master used



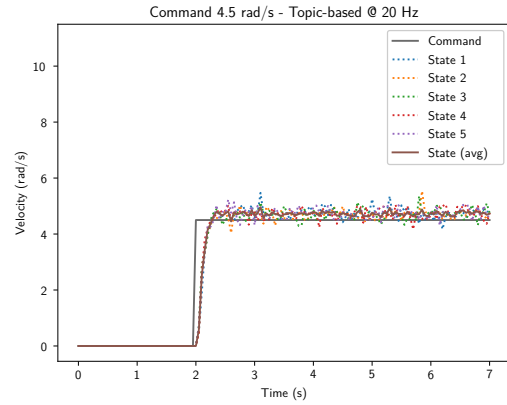
(a) Step response of the original service-based hardware interfaces at 20 Hz with a step input of 3.0 rad s^{-1} while being lifted



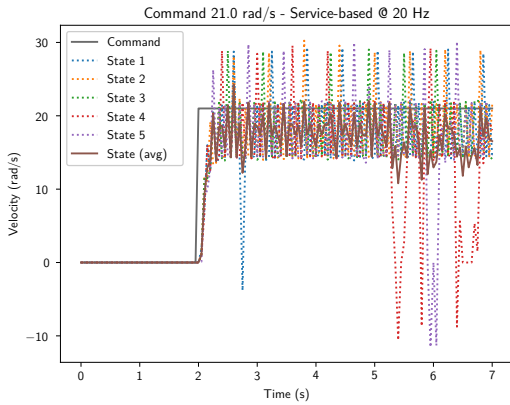
(b) Step response of the proposed topic-based hardware interfaces at 20 Hz with a step input of 3.0 rad s^{-1} while being lifted



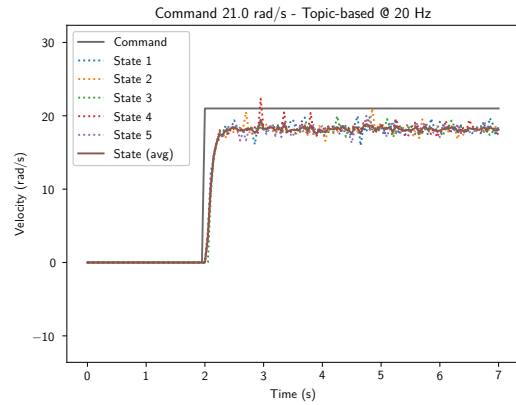
(c) Step response of the original service-based hardware interfaces at 20 Hz with a step input of 4.5 rad s^{-1} while being lifted



(d) Step response of the proposed topic-based hardware interfaces at 20 Hz with a step input of 4.5 rad s^{-1} while being lifted



(e) Step response of the original service-based hardware interfaces at 20 Hz with a step input of 21.0 rad s^{-1} while being lifted



(f) Step response of the proposed topic-based hardware interfaces at 20 Hz with a step input of 21.0 rad s^{-1} while being lifted

Figure 9: Step response of the `ros2_control` hardware interfaces at 20 Hz with step inputs of 3.0 rad s^{-1} , 4.5 rad s^{-1} and 21.0 rad s^{-1} while being lifted. It can be seen that the proposed topic-based setup has significantly less noise than the original service-based setup, and that the original hardware interface reports periodic velocity spikes due to its velocity calculation method. Furthermore, it can also be seen that velocity mapping is not completely accurate, with a clear mismatch of the maximum wheel velocity.

and 16d and Figs. 9e, 9f, 16e and 16f respectively), in which the final velocities are close to 4.83 rad s^{-1} and 18.0 rad s^{-1} respectively, where the topic-based hardware interfaces experience a noise reduction compared to their service-based counterparts.

In addition to this, it can be observed that the maximum wheel velocity is not particularly well tuned. Currently the velocity scaling has been configured to the theoretical maximum velocity of the motor according to their specifications [26], however, these velocities are never reached in the more ideal lifted scenario, as can be seen in Figs. 9e, 9f, 16e and 16f. This can be adjusted for in the proposed hardware interfaces.

Furthermore, all current hardware interfaces have assumed that the pulse-width modulation (PWM) values used to control the motors map linearly to the wheel velocity; however, this is not the case, as can be seen from the results summarized in Table 1. This is most likely related to the method used to control to wheel motors. A MIRTE Master uses 2 L298N Dual full-bridge drivers to control the 4 motors [27]. These L298N modules can expose 3 pins per motor, 2 of these pins control the direction, while the other enables the full-bridge for the specific motor [28], [29]. In the current setup, the enable pins are connected to 5 V directly, while the other pair pins are controlled with 2 PWM pins to control the direction and speed. This is a relatively common way of integrating these motor drivers, however during the low part of the PWM signal, both control pins have an equal logical value, which results in a ‘Fast motor stop’ according to the datasheet as the enable pin is also active [29, Table. 5]. This ‘Fast motor stop’ does not happen long enough to completely stop the motor as a result of the PWM frequency, however, the effect of this is on the motor power in a non-linear way. The alternative usable behavior is to PWM the enable pin of the motor driver, as the datasheet specifies that as a ‘Free running motor stop’ [29, Table. 5], which possibly reduces this effect.

6.2.2 Grounded

In this section, the result of the grounded step response measurement will be discussed. The step response graphs resulting from this test of the front left wheel can be found in Figs. 17 and 10 for 10 and 20 Hz respectively. It is important to note that all tests with step size 3.0 rad s^{-1} have been omitted in this chapter as this command velocity, or actually the command power, is insufficient to move the robot due to the added load of the friction under its own weight.

Table 2: The final wheel velocity averages of the grounded front left wheel recordings. These averages are taken over the interval starting 1 s after the step event until the end of the recording.

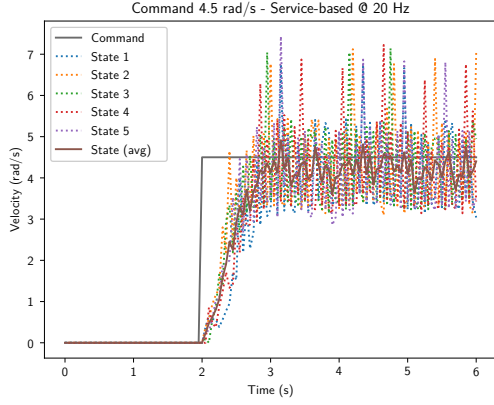
Configuration	Commanded Step Size	
	4.5 rad s^{-1}	21.0 rad s^{-1}
Service Based @ 10 Hz (Original)	4.24 rad s^{-1}	16.5 rad s^{-1}
Topic Based @ 10 Hz	3.87 rad s^{-1}	16.3 rad s^{-1}
Service Based @ 20 Hz	4.23 rad s^{-1}	16.5 rad s^{-1}
Topic Based @ 20 Hz (Proposed)	3.85 rad s^{-1}	16.5 rad s^{-1}
Average	4.05 rad s^{-1}	16.5 rad s^{-1}

In the grounded test case, the final wheel speed is lower than the lifted case as a result of the additional load on the motors due to wheel frictions and the robot’s own weight. This reduces the effective non-linear velocity mapping described in Section 6.2.1. In addition to this, the rise time is significantly higher than in the lifted test. This is expected, as for the grounded tests the inertia of the whole robot needs to be accelerated instead of accelerating only the much smaller inertia of a single wheel.

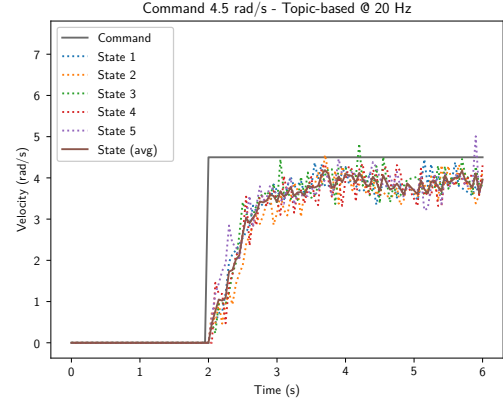
With the added load, the final wheel velocity average drops, compared to the lifted case. The final wheel velocity averages of the front left wheel in the grounded case can be found in Table 2. In this table, it can be seen that the final velocity average is consistent overall configurations in the trials with a command velocity of 21.0 rad s^{-1} . However, the final wheel velocity average in the trials with a command velocity of 4.5 rad s^{-1} is only consistent over the hardware interface type. In this test, the

configurations using the original service-based hardware interface have a final average wheel velocity much closer to the command velocity, with an average over these configurations resulting in 4.24 rad s^{-1} , while the trials using the proposed topic-based hardware interfaces average to only 3.86 rad s^{-1} . This is most likely caused by the noise and artifacts present in the wheel velocity state interface signal in the original hardware interfaces.

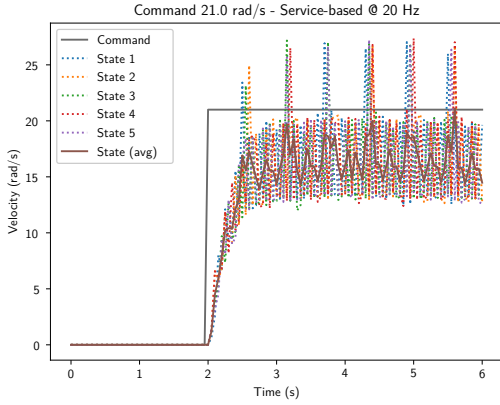
The proposed hardware interfaces experience a lot less noise compared to the original setup, as can be seen in Figs. 10 and 17. However, in trials using the proposed topic-based hardware interfaces, there is still some noise present, as can be seen in Figs. 10b, 10d, 17b and 17d. This can be caused by multiple factors, such as unfavorable CPU scheduling and differences in testing conditions. This could also be caused by inaccuracies in the encoder velocity calculations described by Lima *et al.* [30]. In their work, they introduce a promising method to reduce this noise by compensating for an imperfect magnetic field created by the permanent magnet attached to the motor. This method does require extra hardware-specific calibration to compensate for these unbalanced states [30].



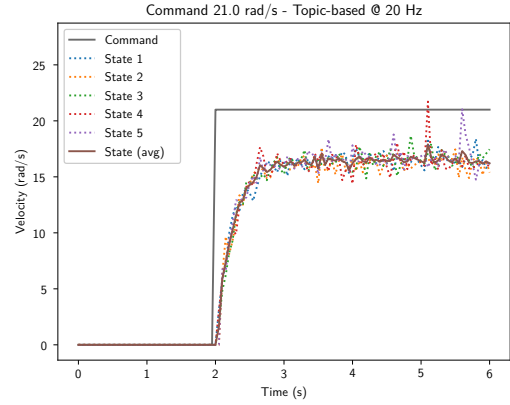
(a) Step response of the original service-based hardware interfaces at 20 Hz with a step input of 4.5 rad s^{-1} while being on the ground



(b) Step response of the proposed topic-based hardware interfaces at 20 Hz with a step input of 4.5 rad s^{-1} while being on the ground

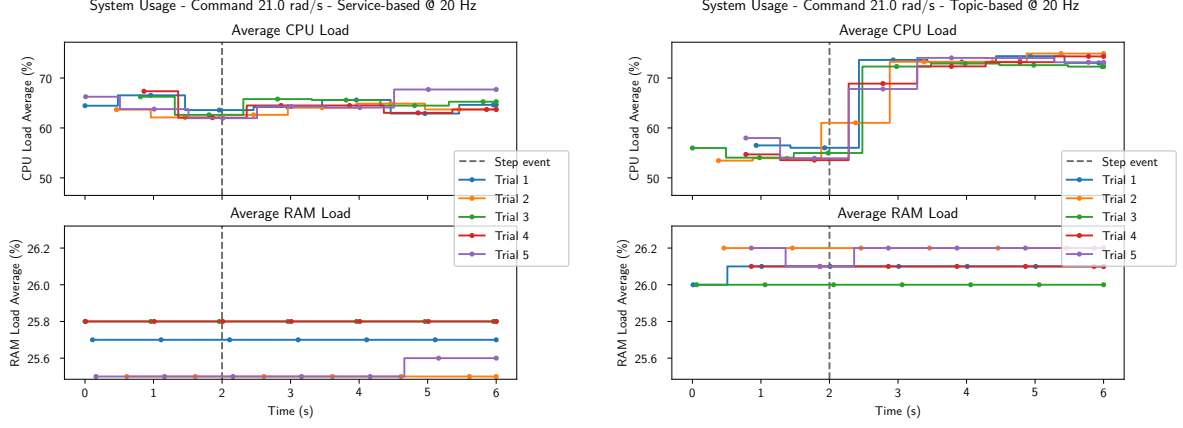


(c) Step response of the original service-based hardware interfaces at 20 Hz with a step input of 21.0 rad s^{-1} while being on the ground



(d) Step response of the proposed topic-based hardware interfaces at 20 Hz with a step input of 21.0 rad s^{-1} while being on the ground

Figure 10: Step response of the `ros2_control` hardware interfaces at 20 Hz with step inputs of 4.5 rad s^{-1} and 21.0 rad s^{-1} while being on the ground. It can be seen that, similarly as in the lifted case (Fig. 9), the proposed topic-based setup has significantly less noise than the original service-based setup, and that the original hardware interface reports periodic velocity spikes due to its velocity calculation method. Furthermore, it can be seen that the additional dynamics of the robot dominate the wheel velocity system response, which can be experienced by the increased rise time.



(a) System resource usage of the MIRTE Master using the original service-based hardware interfaces at 20 Hz during a step response with a step input of 21.0 rad s^{-1} while being on the ground

(b) System resource usage of the MIRTE Master using the proposed topic-based hardware interfaces at 20 Hz during a step response with a step input of 21.0 rad s^{-1} while being on the ground

Figure 11: System resource usage of the MIRTE Master during the step response measurements of the `ros2_control` hardware interfaces at 20 Hz excited with a step input of 21.0 rad s^{-1} while being on the ground. It can be seen that the proposed topic-based setup has a lower CPU utilization when the robot is idle compared to the original service-based setup. It can also be seen that the RAM utilization is not impacted significantly.

6.2.3 System resources

In this section, the system resource utilization of the grounded step response measurement will be discussed. The system resource utilization graphs can be found in Figs. 18 and 11 for 10 and 20 Hz respectively. Only the resource usage graphs of the tests with a command velocity of 21.0 rad s^{-1} are shown, as the results are mostly invariant with respect to the commanded velocity.

With the original control setup, the CPU usage of the full system remains relatively constant across each trial, averaging around 60% and 65% for 10 and 20 Hz respectively, as can be seen in Figs. 11a and 18a. The proposed topic-based system exhibits a different behavior, where the system usage is lower when the robot is idle compared to when the robot is actively moving (see Figs. 11b and 18b). The CPU usage during the idle part of the measurements is approximately 50% and 55% for 10 and 20 Hz respectively, while the CPU usage during locomotion is 67.5% and 72.5% respectively. This can be beneficial in certain cases, for example, when the robot is scanning its environment and running an advanced detection algorithm to find its next goal. During which the robot's locomotion system is likely to be idle in order to receive clearer sensor data from the cameras and the LiDAR. This behavior can potentially increase the battery life of the robot, as the robot uses significantly less computational resources when its idle.

The RAM usage of the robot during all tests has remained mostly stable across all configurations, which is to be expected as the amount of transferred data has not changed significantly.

It is important to note that both the RAM and CPU usage are most likely effected by the recording setup. This has happened since under normal operation in the default configuration none of the recorded data streams, e.g. the ROS topics and ROS services, would leave the robot's SBC. However, to not influence the SBC of the MIRTE Master too much with disk and compression operations, a carefully balanced decision was made to run the recording and test-related nodes (except for the custom `ros2_control` controllers) on a different computer connected via Ethernet. This has the benefit of offloading the relatively expensive data storage and compression to a more powerful processor, at the cost of extra network transport overhead on the robot's SBC.

7 Conclusion and Discussion

The goal of this thesis was to improve the wheel motors' system response of the MIRTE Master robot by adapting its control architecture. This has been achieved by creating a new `ros2_control` setup with modular `ros2_control` hardware interfaces, which only communicates the changes with the `telemetry` hardware communication node using ROS topics. These new hardware interfaces also use a more direct way of calculating the wheel velocity in a frequency-independent way. This is accompanied by a number of `telemetry` improvements to allow for update-only encoder readings and command synchronization support for the yet-to-be released hardware revision.

Due to these optimizations, the base control system can now be configured to run at 20 Hz without increasing the system resource usage compared to the original, while the CPU usage at idle has decreased. These hardware interfaces can also be easily configured with the actual hardware parameters, which allows for better tuning for the individual hardware instances. Furthermore, there is a significant noise reduction in the velocity signal as a result of the improved calculation method. The coarse artificial discretization has been removed, as it was no longer a necessary optimization. All this has resulted in the gain and phase margin increasing from 2.6 dB and 70° to 6 dB and 135° respectively.

Future Work

A lot of work has already been done on the improved `ros2_control` setup. However, there were more possible improvements identified, which were too much work to implement and test in this thesis. Therefore, there are still points where the work can be improved and expanded upon. These points will be discussed here.

Microcontroller-based data timestamping

In the current implementation, the sensor data is timestamped when it is published as a ROS message. This, however, has several issues due to delay between the measurement and ROS publication, 'out-of-order' messages caused by Network Time Protocol (NTP) synchronization, and clock drift on the SBC [31]. There are multiple ways to improve this situation, the simplest being using a monotonically increasing or steady ROS clock to generate the timestamps and also timestamping the messages earlier. The best solution, however, is to send the time according to the clock on the microcontroller with the sensor data. This would be more stable, as the clocks on microcontrollers are often based on highly stable oscillating crystals. Furthermore, this timestamp would actually be related to the time the measurement was taken. Using these timestamps would result in an even larger noise reduction on the wheel velocity. This would however require a larger effort to implement clock rate detection, to be able to relate the microcontroller's clock to the system clock of the SBC.

Only microcontroller-driven publications for `telemetry`

In the original ROS 1 implementation of `telemetry`, it was determined that the ROS node couldn't keep up with a constant datastream. Therefore it was decided to only send updates from the microcontroller to the SBC, on which `telemetry` would publish the last received data at a fixed rate. This was implemented in a similar fashion in the new C++ `telemetry` implementation. However, this combined with the use of a multithreaded executor results in a relatively high CPU usage. This can also cause two fast repeated messages, with one of them containing the old data. This can then throw off the recipient of those messages. An alternative that could be explored is to send the sensor data every iteration on the microcontroller and then only publish these messages from ROS when they are received by `telemetry`. This would theoretically result in a much more consistent message frequency while preventing unnecessary rapid publication and reducing the CPU usage.

Reduce the use of multithreading in `telemetry` for improved determinism

The original ROS 1 implementation had significant responsiveness issues on the ROS communication side. This was most likely caused by the performance overhead of the Python implementation combined with some (now-solved) bugs/inefficiencies in the microcontroller firmware related to MIRTE Master specific features/devices. These most significant issues included the original method of writing to the

OLED screen, which gets periodically updated with status information on the MIRTE Master and the communication with the HiWonder serial servos. These servos have their UART transmit (TX) and receive (RX) lines connected in such a way that the servos relay everything from RX to TX. These messages have to be deduplicated on the microcontroller, which has a significant computational cost as the servos UART bus is shared with the USB connection to the SBC.

The effects of these performance issues in the ROS 1 version were observed and taken into account when creating the ROS 2 C++ version. However, the causes have been identified later, resulting in the designs for `tmx_cpp` and `telemetry` compensating for limitations of the original version, which might not be applicable anymore. This entails that the motivation for the use of a worker thread pool to process the microcontroller messages and the use of `MultiThreadedExecutor` might be invalid. This can cause issues as context switching in a multithreaded workload is not without a cost, while the positive impact might be limited.

Composing `telemetry` and `ros2_control` to enable the use of intra-process communication

ROS 2 does not have a single transport system like ROS 1, but run-time loadable RMW implementations. These RMW implementations, such as `rmw_fastrtps_cpp` and `rmw_zendoh_cpp`, serialize and deserialize the exchanged data. The combination of serialization, sending, receiving, and deserializing can introduce significant variable delays in the message pipeline, making these methods non-deterministic.

There are two methods possible to make this transport more consistent. The most limited but easiest option is enabling inter-process communication. This allows messages between nodes on the same machine to be transported purely in memory by pointer passing instead of via (local) network traffic. This needs to be implemented by the underlying transport library used by a specific RMW and still serializes the data being transported. As the specifics of this feature are RMW dependent and often still use multithreading, the improvements from this method are limited for this usecase.

The alternative is using intra-process communication, which completely skips the RMW layer and its serialization [5], [32], [33]. While intra-process communication has lower latency, it is limited to nodes in the same `rclcpp::Context`[34]. Which can only be achieved by embedding the desired nodes in a single executable or by using composition, which allows these node components to be loaded into a single context at runtime. This also enables the nodes to share a (single-threaded) executor, improving determinism. However, the `telemetry` node and `ros2_control` controller manager are not setup for composition and therefore require either refactoring to enable composition or creating a custom executable. These options have only been explored conceptually for this work due to the amount of work required for this optimization.

Verify message storage lock behavior in proposed encoder hardware interface

Currently, the proposed encoder hardware interface shows strange artifacts at higher frequencies. It seems to be caused by not receiving a new encoder position on time, which initially was written of as an effect of communication jitter and unfavorable CPU scheduling. However, later I realized that this effect could also be caused by using a partially updated new and old message. The used message storage has a locking mechanism, however in the read function of this hardware interface, it is locked twice sequentially to read the new and the old message. This lock could be taken in between these locks by the encoder message callback, which updates the stored messages. It needs to be checked if this zero event behavior disappears if both messages were to be retrieved at once, as this would enable even higher control frequencies.

Acronyms

ROS	Robot Operating System
RMW	ROS 2 middleware
REP	ROS Enhancement Proposal
LiDAR	light detection and ranging
MIRTE	MIRTE, an Inspiring Robot for Technology Education
TSN	Time-Sensitive Networking
LTS	long-term support
CPU	central processing unit
RAM	random-access memory
SBC	single-board computer
USB	Universal Serial Bus
PWM	pulse-width modulation
UART	universal asynchronous receiver-transmitter
TX	transmit
RX	receive
PID	proportional–integral–derivative

References

- [1] M. Klomp, K. Araffa and G. N. Saunders-Smiths, ‘MIRTE: An affordable, open, mobile robot education platform,’ in *Proceedings of the 52nd Annual Conference of SEFI*, (Lausanne, Switzerland, 2nd–5th Sep. 2024), Zenodo, 2024. DOI: [10.5281/zenodo.14256821](https://doi.org/10.5281/zenodo.14256821).
- [2] D. Niculae, C. Plaisanu and D. Bistriceanu, ‘Sampling jitter compensation for numeric PID controllers,’ in *Proceedings of 2008 IEEE International Conference on Automation, Quality and Testing, Robotics, TOME II*, (Cluj-Napoca, Romania, 22nd–25th May 2008), 3 vols., vol. 2, 2008, pp. 100–104, ISBN: 978-1-4244-2576-1. DOI: [10.1109/AQTR.2008.4588802](https://doi.org/10.1109/AQTR.2008.4588802). IEEE Cat. No. [CFP08AQT-PRT](https://doi.org/10.1109/AQTR.2008.4588802).
- [3] C. E. Shannon, ‘Communication in the presence of noise,’ *Proceedings of the Institute of Radio Engineers*, vol. 37, no. 1, pp. 10–21, 31st Jan. 1949, ISSN: 2162-6634. DOI: [10.1109/JRPROC.1949.232969](https://doi.org/10.1109/JRPROC.1949.232969).
- [4] M. Quigley *et al.*, ‘ROS: An open-source Robot Operating System,’ in *ICRA Workshop on Open Source Software*, 2009.
- [5] S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall, ‘Robot Operating System 2: Design, architecture, and uses in the wild,’ *Science Robotics*, vol. 7, no. 66, eabm6074, 11th May 2022, ISSN: 2470-9476. DOI: [10.1126/scirobotics.abm6074](https://doi.org/10.1126/scirobotics.abm6074).
- [6] S. Chitta *et al.*, ‘`ros_control`: A generic and simple control framework for ROS,’ *Journal of Open Source Software*, vol. 2, no. 20, 456, 2017, ISSN: 2475-9066. DOI: [10.21105/joss.00456](https://doi.org/10.21105/joss.00456).
- [7] A. Yorinks. ‘Telemetry and Telemetry-AIO User’s Guide.’ version 1.45. (9th Nov. 2020), [Online]. Available: <https://mryslab.github.io/telemetry/> (visited on 16/12/2025).
- [8] C. Tezel and G. Bayar, ‘Theoretical and experimental investigation of variable contact forces on the rollers of a mecanum wheeled mobile robot,’ *The International Journal of Robotics Research*, vol. 43, no. 8, pp. 1208–1227, Jul. 2024, ISSN: 0278-3649. DOI: [10.1177/02783649241228607](https://doi.org/10.1177/02783649241228607).
- [9] D. S. Alhanov and V. I. Rubtsov, ‘Development of the laboratory work: “modeling of a mobile robot on mecanum wheels kinematics”,’ *ITM Web of Conferences*, vol. 35, 04001, 2020, ISSN: 2271-2097. DOI: [10.1051/itmconf/20203504001](https://doi.org/10.1051/itmconf/20203504001).

- [10] N. T. Thanh, N. T. Long, T. T. K. Ly, N. H. Thai and H. Thien, ‘Dynamics Simulation of a Mecanum-Wheeled Omnidirectional Mobile Robots,’ in *Advances in Engineering Research and Application, Proceedings of the International Conference on Engineering Research and Applications, ICERA 2023*, D. C. Nguyen, D. T. Hai, N. P. Vu, B. T. Long, H. Puta and K.-U. Sattler, Eds., vol. 943, Springer, 2024, pp. 435–449, ISBN: 978-3-031-62237-3. DOI: [10.1007/978-3-031-62238-0_46](https://doi.org/10.1007/978-3-031-62238-0_46).
- [11] C. He, D. Wu, K. Chen, F. Liu and N. Fan, ‘Analysis of the Mecanum wheel arrangement of an omnidirectional vehicle,’ *Proceedings of the Institution of Mechanical Engineers, Part C*, vol. 233, no. 15, pp. 5329–5340, Aug. 2019, ISSN: 0954-4062. DOI: [10.1177/0954406219843568](https://doi.org/10.1177/0954406219843568).
- [12] A. Cervin, ‘Improved scheduling of control tasks,’ in *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS’99*, York, UK, 1999, pp. 4–10. DOI: [10.1109/EMRTS.1999.777444](https://doi.org/10.1109/EMRTS.1999.777444).
- [13] M. Lluesma, A. Cervin, P. Balbastre, I. Ripoll and A. Crespo, ‘Jitter Evaluation of Real-Time Control Systems,’ in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, (Sydney, NSW, Australia, 16th–18th Aug. 2006), 2006, pp. 257–260, ISBN: 0-7695-2676-4. DOI: [10.1109/RTCSA.2006.41](https://doi.org/10.1109/RTCSA.2006.41).
- [14] A. Dubey, G. Karsai and S. Abdelwahed, ‘Compensating for timing jitter in computing systems with general-purpose operating systems,’ in *Proceedings of the 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, (Tokyo, Japan, 17th–20th Mar. 2009), IEEE, 2009, pp. 55–62, ISBN: 978-0-7695-3573-9. DOI: [10.1109/ISORC.2009.28](https://doi.org/10.1109/ISORC.2009.28).
- [15] Y. L. Ye, Z. G. Nie, X. J. Liu, F. G. Xie, Z. H. Li and P. Li, ‘ROS2 Real-time Performance Optimization and Evaluation,’ *Chinese Journal of Mechanical Engineering*, vol. 36, no. 144, 2023, ISSN: 2192-8258. DOI: [10.1186/s10033-023-00976-5](https://doi.org/10.1186/s10033-023-00976-5).
- [16] L. Puck *et al.*, ‘Performance Evaluation of Real-Time ROS2 Robotic Control in a Time-Synchronized Distributed Network,’ in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE 2021)*, (Lyon, France, 23rd–27th Aug. 2021), IEEE, 2021, pp. 1670–1676, ISBN: 978-1-6654-1873-7. DOI: [10.1109/CASE49439.2021.9551447](https://doi.org/10.1109/CASE49439.2021.9551447).
- [17] Dynamixel Community Contributors, *dynamixel_hardware*, 2021. Available: [GitHub.com:dynamixel-community/dynamixel_hardware](https://github.com/dynamixel-community/dynamixel_hardware) at tag 0.5.0.
- [18] OUXT-Polaris Contributors, *nmea_hardware_interface*, 2021. Available: [GitHub.com:OUXT-Polaris/nmea_hardware_interface](https://github.com/OUXT-Polaris/nmea_hardware_interface) at commit 1d49893.
- [19] G. Bukka, D. Štogl and Clearpath Robotics, *clearpath_mecanum_drive_controller*, 2023. Available: [GitHub.com:clearpathrobotics/clearpath_mecanum_drive_controller](https://github.com/clearpathrobotics/clearpath_mecanum_drive_controller) at tag 0.1.1.
- [20] G. Bukka and D. Štogl, *mecanum_drive_controller*, 2023. Available: [GitHub.com:ros-controls/ros2_controllers](https://github.com/ros-controls/ros2_controllers) at tag 4.37.0.
- [21] Open Robotics, *Configure service introspection*, ROS 2 Documentation: Jazzy documentation, version Jazzy. [Online]. Available: <https://docs.ros.org/en/jazzy/Tutorials/Demos/Service-Introspection.html> (visited on 26/01/2026).
- [22] ROS 2 rosbag2 Contributors, *rosbag2*, 2018. Available: [GitHub.com:ros2/rosbag2](https://github.com/ros2/rosbag2) at tag 0.26.9.
- [23] T. Foote, *Diagnostic System for Robots Running ROS*, Open Robotics, 27th Nov. 2010. [Online]. Available: <https://reps.openrobotics.org/rep-0107/> (visited on 04/02/2026).
- [24] ROS 2 rcl Contributors, *rcl*, 2015. Available: [GitHub.com:ros2/rcl](https://github.com/ros2/rcl) at tag 9.2.9.
- [25] ROS 2 rclcpp Contributors, *rclcpp*, 2014. Available: [GitHub.com:ros2/rclcpp](https://github.com/ros2/rclcpp) at tag 28.1.16.
- [26] Micro DC Gear Motor Factory Store. ‘37 mm Diameter Low Speed Silent 12 V 24 V Geared Motor Eccentric Shaft Gearmotor With CPR=44 Speed Measurement Encoder Gear Motor,’ AliExpress. (4th Aug. 2025), [Online]. Available: <http://aliexpress.com/item/1005005021902364.html> (visited on 04/08/2025).

- [27] MIRTE team. ‘Hardware specs, MIRTE Documentation v0.1 documentation.’ (4th Aug. 2025), [Online]. Available: https://docs.mirte.org/develop/doc/robots/mirte_master/hardware_specs.html (visited on 04/08/2025).
- [28] STMicroelectronics. ‘L298 | Product - STMicroelectronics.’ (2026), [Online]. Available: <https://www.st.com/en/motor-drivers/l298.html> (visited on 20/01/2026).
- [29] STMicroelectronics, *Datasheet - L298 - Dual full-bridge driver*, version 5, 6th Oct. 2023. [Online]. Available: <https://www.st.com/resource/en/datasheet/l298.pdf> (visited on 20/01/2026).
- [30] J. Lima, V. H. Pinto, A. P. Moreira and P. Costa, ‘Improving Incremental Encoder Measurement: Variable Acquisition Window and Quadrature Phase Compensation to Minimize Acquisition Errors,’ in *2022 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, (Santa Maria da Feira, Portugal, 29th–30th Apr. 2022), L. Louro, P. Fonseca, P. Neto and R. Ventura, Eds., IEEE, 2022, pp. 118–123, ISBN: 978-1-6654-8217-2. DOI: [10.1109/ICARSC55462.2022.9784766](https://doi.org/10.1109/ICARSC55462.2022.9784766).
- [31] B. Wittenmark, B. Bastian and J. Nilsson, ‘Analysis of time delays in synchronous and asynchronous control loops,’ in *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, vol. 1, Tampa, Florida, USA: IEEE, 1998, pp. 283–288. DOI: [10.1109/CDC.1998.760685](https://doi.org/10.1109/CDC.1998.760685).
- [32] A. Soragna, J. Oxoby and D. Goel, ‘Intra-process Communications in ROS 2,’ *ROS 2 Design*, 30th Mar. 2020. [Online]. Available: https://design.ros2.org/articles/intraprocess_communications.html (visited on 15/01/2026).
- [33] S. Macenski, A. Soragna, M. Carroll and Z. Ge, ‘Impact of ROS 2 Node Composition in Robotic Systems,’ *IEEE Robotics and Autonomous Letters (RA-L)*, vol. 8, no. 7, pp. 3996–4003, 24th May 2023, ISSN: 2377-3766. DOI: [10.1109/LRA.2023.3279614](https://doi.org/10.1109/LRA.2023.3279614).
- [34] rclcpp Contributors, *Class IntraProcessManager*, rclcpp: Jazzy 28.1.16 documentation, version 28.1.16, 21st Oct. 2019. [Online]. Available: https://docs.ros.org/en/jazzy/p/rclcpp/generated/classrclcpp_1_1experimental_1_1IntraProcessManager.html (visited on 04/02/2026).
- [35] MIRTE team, *Telemetrix4RpiPico*, 2021. Available: [GitHub.com:mirte-robot/Telemetrix4RpiPico](https://github.com/mirte-robot/Telemetrix4RpiPico) at commit 83d11dd.
- [36] J. Smoot. ‘What’s the Difference Between an Incremental Encoder’s PPR, CPR, and LPR?’ Same Sky. (6th Jun. 2017), [Online]. Available: <https://www.sameskydevices.com/blog/what-is-encoder-ppr-cpr-and-lpr> (visited on 04/08/2025).
- [37] Python Packaging Authority, *Externally Managed Environments*, Python Packaging User Guide, 6th Mar. 2023. [Online]. Available: <https://packaging.python.org/en/latest/specifications/externally-managed-environments/> (visited on 16/02/2026).

A Encoder Tick Verification

In this appendix, it will be verified that no encoder ticks are missed by means of an analytical calculation. This is necessary as there were suspicions that this was not case, due to the polling nature of the encoder readings. If this is not the case, then the wheel velocity must be measured externally to in all further tests, while an alternative encoder reading solution, such as the method described by Lima *et al.* [30].

In the original ROS implementation of MIRTE’s hardware communication module `mirte_telemetrix_cpp`⁹, the encoder state is retrieved using a timer running at a scanning frequency (f_c) of 10 kHz. Often encoder values are updated using interrupts, ensuring no encoder ticks are missed. However, this alternative approach using a timer allows for more flexibility, even functioning on microcontrollers with a very limited number of interrupt-capable pins.

Table 3: Encoder Tick Verification Symbols

Symbol	Description
ω_m	The rotational speed of the motor/encoder
ω_w	The rotational speed of the wheel (after the gearbox)
$f_{s,\max}$	The number of ticks of the encoder per second at the maxixum speed
c_w	The number of ticks of the encoder per wheel revolution
$T_{s,\min}$	The minimum time between encoder ticks
T_c	The interval between encoder state observations

Table 4: Encoder Tick Verification Constants

Symbol	Value	Description
r	30^{-1}	The gearbox ratio ($\frac{\omega_w}{\omega_m}$)*
$\omega_{w,\max}$	200 min^{-1}	The maximum motor speed (in revolutions per minute (RPM))*
c_m	44	The number of encoder ticks/states the encoder reports per motor revolution [36]*
f_c	10 kHz	Encoder scanning frequency [35]

* Value has been retrieved from the sellers listing [26], which was provided by MIRTE team [27].

In this section, an analysis of this system will be conducted. All symbols and constants are defined in Tables 3 and 4. In order for this system to not miss any controller ticks during normal (one directional operation), Eq. (1) must hold, which states that the time between state checks (T_c) must be smaller than the minimum time between state transitions ($T_{s,\min}$).

$$T_c < T_{s,\min} \quad (1)$$

The time between state checks (T_c) can be found using Eq. (2). The duration between state checks is 0.1 ms.

$$T_c = \frac{1}{f_c} = 10^{-4} \text{ s} \quad (2)$$

The minimum time between state transitions ($T_{s,\min}$) occurs when the the states change at the maximum frequency ($f_{s,\max}$). These are related by Eq. (3).

$$T_{s,\min} = \frac{1}{f_{s,\max}} \quad (3)$$

This maximum state change frequency ($f_{s,\max}$) is determined by the number of states per revolution and the amount of revolutions per second. This can be either on the wheel side (using c_w and $\omega_{w,\max}$) or on the motor side (using c_m and $\omega_{m,\max}$). This relation is described in Eq. (4).

⁹Using their `tmx_cpp` library to talk to compatible microcontroller firmwares, in our case with the Raspberry Pi Pico running the `Telemetrix4RpiPico` firmware[35].

$$f_{s,\max} = c_w \omega_{w,\max} = c_m \omega_{m,\max} \quad (4)$$

The wheel and motor are connected with a gearbox with transmission ratio r . The wheel and motor speed are correlated using Eq. (5).

$$\omega_{m,\max} = \frac{\omega_{w,\max}}{r} \quad (5)$$

Combining Eqs. (3) to (5), an equation for $T_{s,\min}$ can be derived. This equation can be found in Eq. (6), and the minimum time between state transitions is 0.227 ms.

$$T_{s,\min} = \frac{r}{c_m \omega_{w,\max}} = 2.27 \times 10^{-4} \text{ s} \quad (6)$$

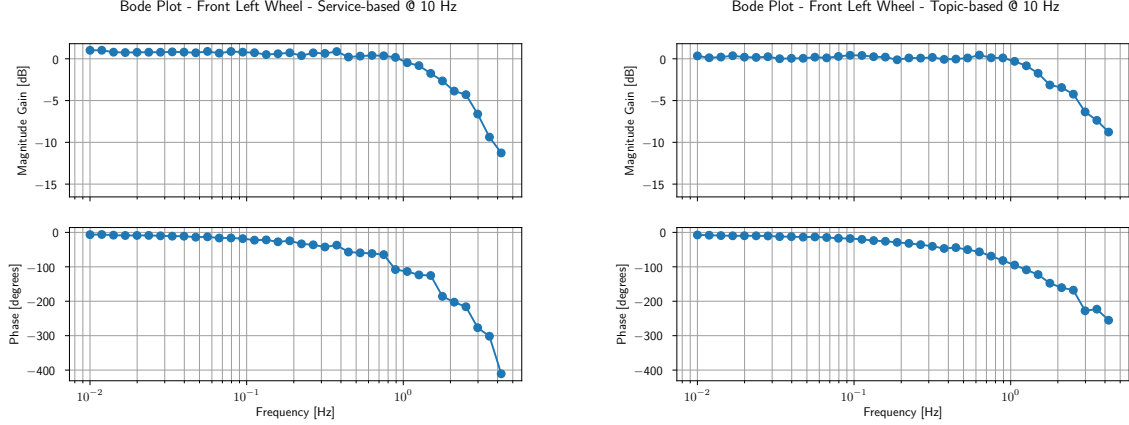
Given the values determined for T_c and $T_{s,\min}$ (See Eqs. (2) and (6) respectively), it can be concluded that Eq. (1) holds, meaning that no encoder ticks can be missed during normal operation.

B 10 Hz Measurement Data

This appendix contains the graphs of the measurement results of the original and proposed `ros2_control` setups at the original frequency of 10 Hz for the front left wheel.

B.1 Frequency Response - No Offset

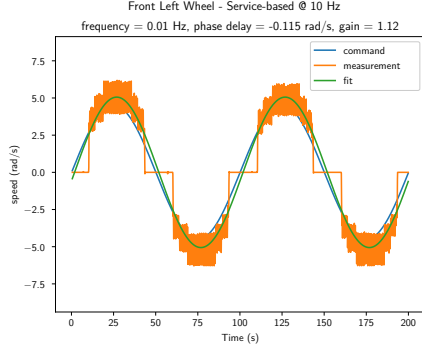
This section contains Figs. 12 and 13, which are the frequency response graphs measured without an offset for the front left wheel using the original and proposed `ros2_control` setups running at 10 Hz.



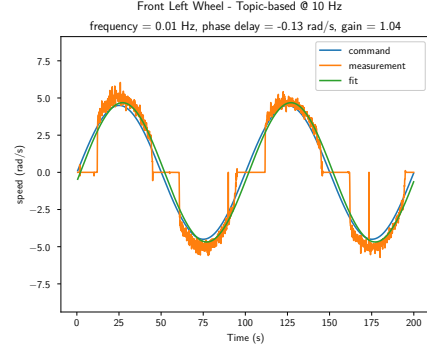
(a) Frequency response of the original service-based hardware interfaces at 10 Hz using a sinusoid around zero. From this graph, it can be found that the gain and phase margin are 2.6 dB and 70° in this configuration.

(b) Frequency response of the proposed topic-based hardware interfaces at 10 Hz using a sinusoid around zero. From this graph, it can be found that the gain and phase margin are 6.1 dB and 100° in this configuration.

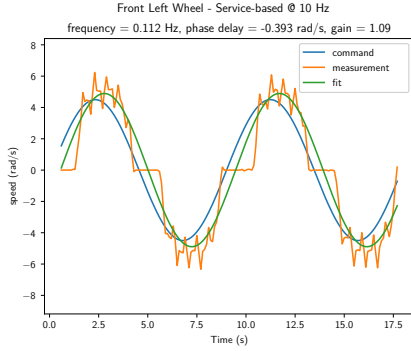
Figure 12: Frequency response of the hardware interfaces generated using a sinusoid around zero while being lifted with a `ros2_control` controller frequency of 10 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 10 Hz, the maximum measurable frequency is limited to below 5 Hz as described in the Nyquist-Shanon theorem [3]. In the subfigures, it can be seen that the proposed topic-based `ros2_control` setup has a larger phase and gain margin compared to the original service-based setup.



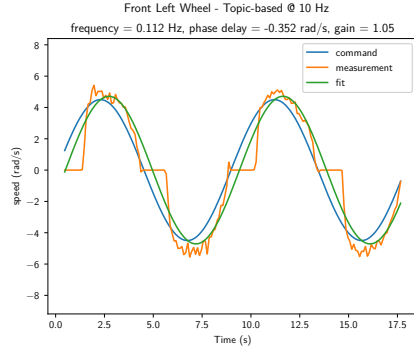
(a) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 0.01 Hz



(b) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 0.01 Hz

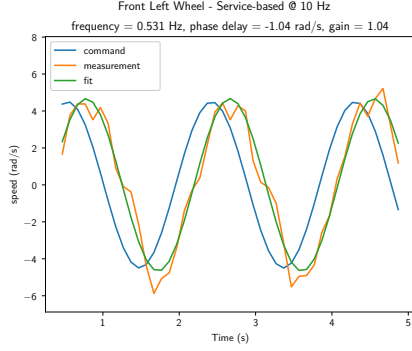


(c) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 0.112 Hz

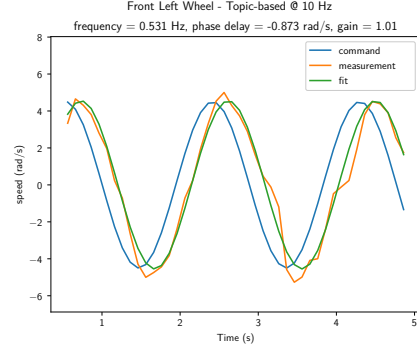


(d) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 0.112 Hz

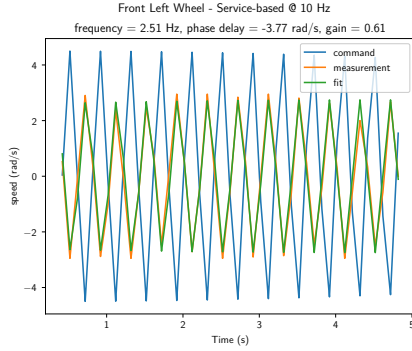
Figure 13: The individual trial fits for the frequency response measurements generated using a sinusoid around zero while being lifted with a `ros2_control` controller frequency of 10 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 10 Hz, the maximum measurable frequency is limited to below 5 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency and amplitude as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 12. In these subfigures it can be seen that the proposed topic-based setup has a finer command discretization and a smaller command deadzone compared to the original service-based setup.



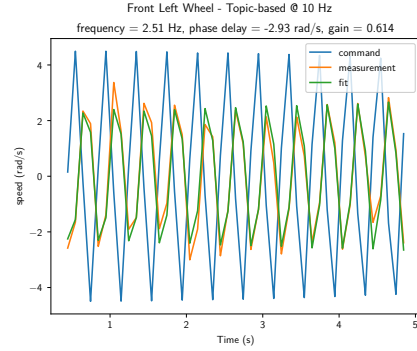
(e) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 0.531 Hz



(f) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 0.531 Hz



(g) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 2.51 Hz

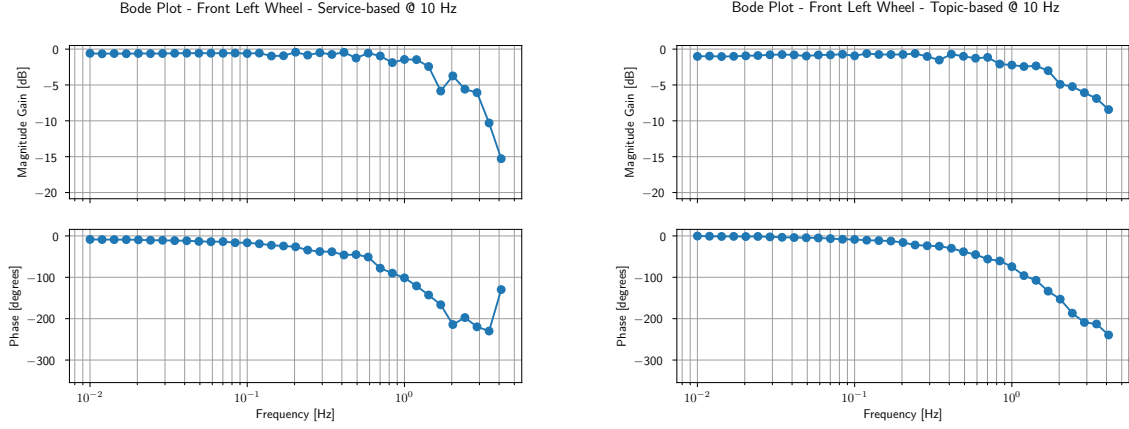


(h) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with a sinusoid with a frequency of 2.51 Hz

Figure 13: The individual trial fits for the frequency response measurements generated using a sinusoid around zero while being lifted with a `ros2_control` controller frequency of 10 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 10 Hz, the maximum measurable frequency is limited to below 5 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency and amplitude as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 12. In these subfigures it can be seen that the proposed topic-based setup has a finer command discretization and a smaller command deadzone compared to the original service-based setup. (cont.).

B.2 Frequency Response - Offset

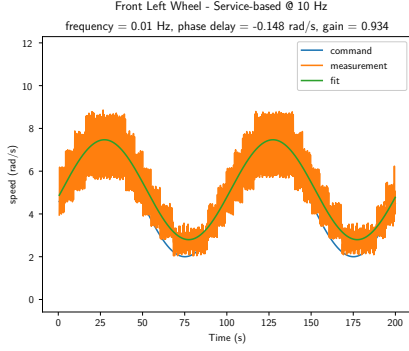
This section contains Figs. 14 and 15, which are the frequency response graphs measured with an offset of 4.5 rad s^{-1} for the front left wheel using the original and proposed `ros2_control` setups running at 10 Hz.



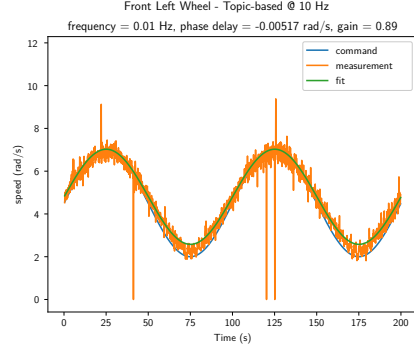
(a) Frequency response of the original service-based hardware interfaces at 10 Hz using an offset sinusoid. From this graph, it can be found that the gain margin in this configuration is 3.5 dB. The phase margin cannot be determined as the graph is always below 0 dB in the measured range.

(b) Frequency response of the proposed topic-based hardware interfaces at 10 Hz using an offset sinusoid. From this graph, it can be found that the gain margin in this configuration is 5.2 dB. The phase margin cannot be determined as the graph is always below 0 dB in the measured range.

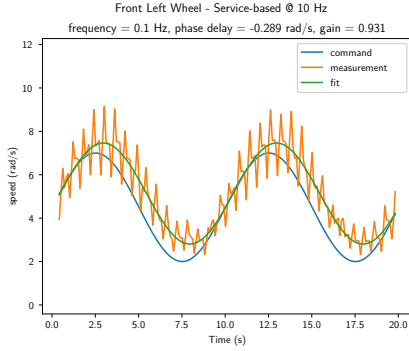
Figure 14: Frequency response of the hardware interfaces generated using a sinusoid with an offset of 4.5 rad s^{-1} while being lifted with a `ros2_control` controller frequency of 10 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 10 Hz, the maximum measurable frequency is limited to below 5 Hz as described in the Nyquist-Shanon theorem [3]. In the subfigures, it can be seen that the proposed `ros2_control` setup has a higher gain margin compared to the original setup.



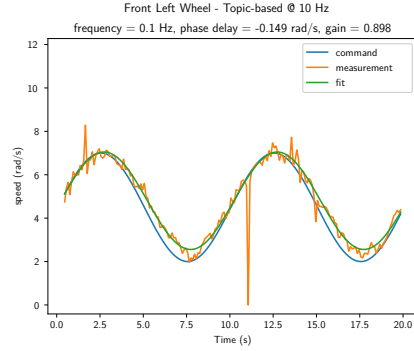
(a) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 0.01 Hz



(b) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 0.01 Hz

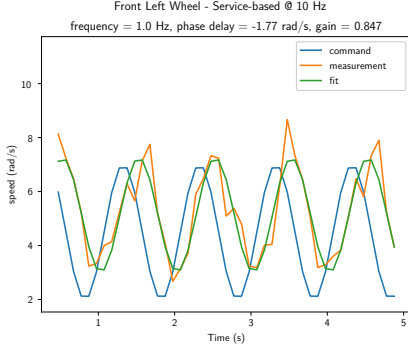


(c) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 0.1 Hz

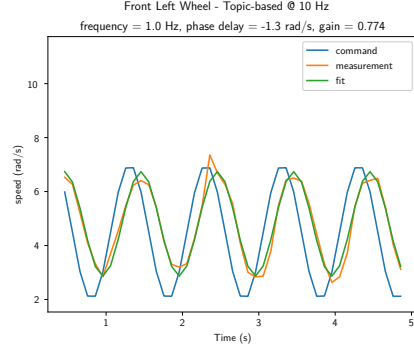


(d) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 0.1 Hz

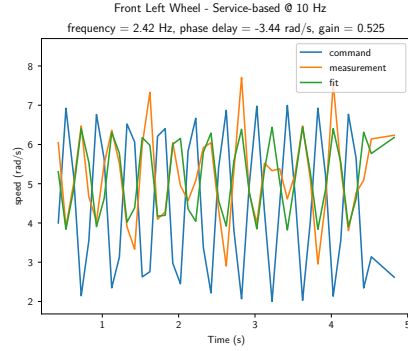
Figure 15: The individual trial fits for the frequency response measurements generated using a sinusoid with an offset of 4.5 rad s^{-1} while being lifted with a `ros2_control` controller frequency of 10 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 10 Hz, the maximum measurable frequency is limited to below 5 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency, amplitude and offset as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 14. In these subfigures, it can be seen that the proposed topic-based setup has significantly less noise in the velocity state signal compared to the original service-based setup.



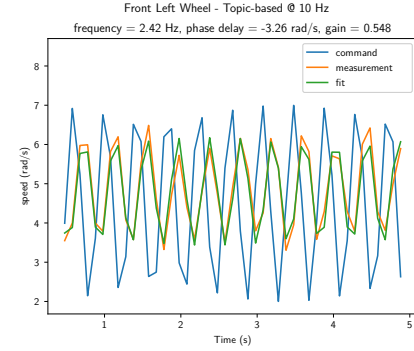
(e) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 1.0 Hz



(f) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 1.0 Hz



(g) Original service-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 2.42 Hz

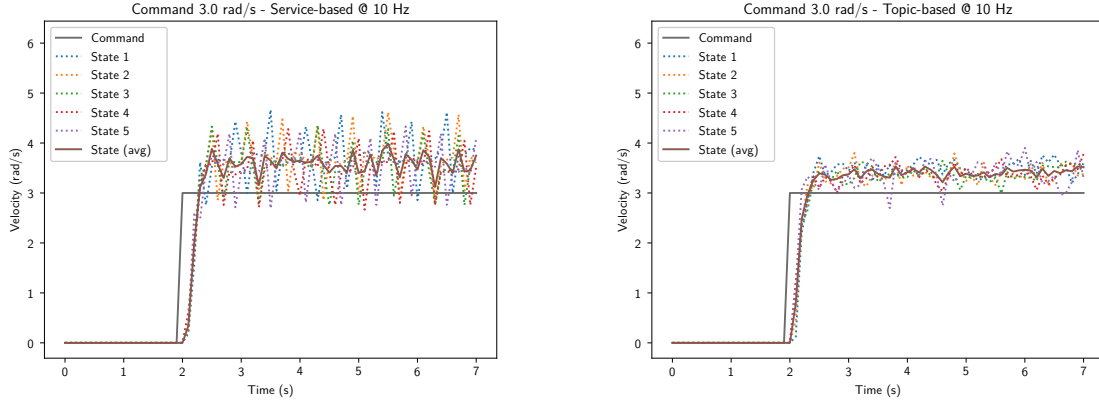


(h) Proposed topic-based `ros2_control` hardware interfaces running at 10 Hz, driven with an offset sinusoid with a frequency of 2.42 Hz

Figure 15: The individual trial fits for the frequency response measurements generated using a sinusoid with an offset of 4.5 rad s^{-1} while being lifted with a `ros2_control` controller frequency of 10 Hz for the Mecanum drive-related components. Due to the `ros2_control` frequency being 10 Hz, the maximum measurable frequency is limited to below 5 Hz as described in the Nyquist-Shanon theorem [3]. Each subfigure shows the reference and measured signals of the front left wheel. There is also a sinusoid fit on the measured data, which is a sinusoid at the same frequency, amplitude and offset as the reference signal, with the phase delay and magnitude gain as a free parameter. The data extracted from this fitted sinusoid is used to generate the bode plots in Fig. 14. In these subfigures, it can be seen that the proposed topic-based setup has significantly less noise in the velocity state signal compared to the original service-based setup. (cont.).

B.3 Step Response - Lifted

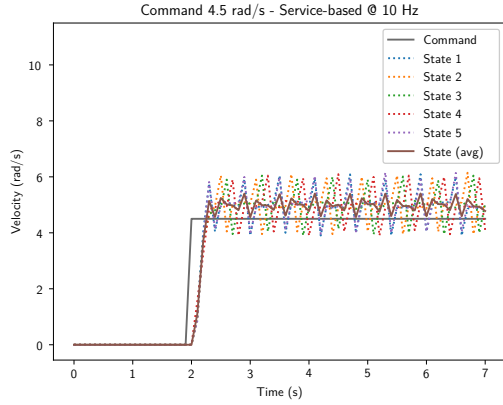
This section contains Fig. 16, which is the step response graphs measured while being lifted for the front left wheel using the original and proposed `ros2_control` setups running at 10 Hz.



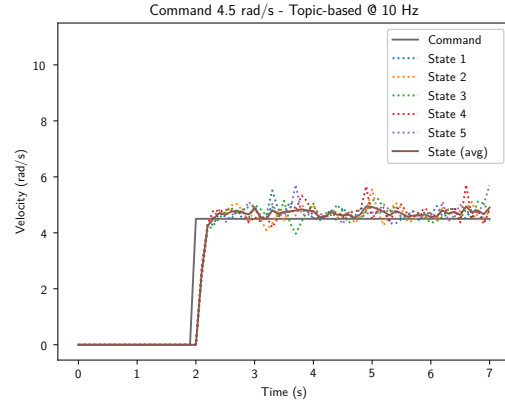
(a) Step response of the original service-based hardware interfaces at 10 Hz with a step input of 3.0 rad s^{-1} while being lifted

(b) Step response of the proposed topic-based hardware interfaces at 10 Hz with a step input of 3.0 rad s^{-1} while being lifted

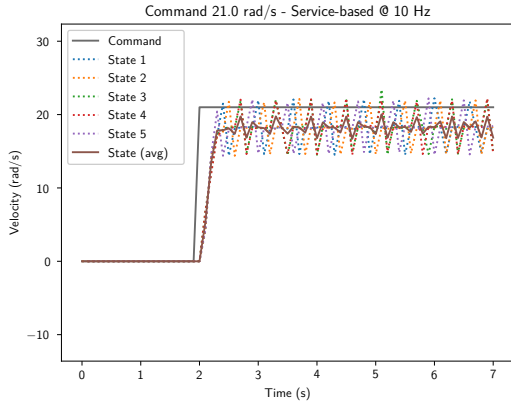
Figure 16: Step response of the `ros2_control` hardware interfaces at 10 Hz with step inputs of 3.0 rad s^{-1} , 4.5 rad s^{-1} and 21.0 rad s^{-1} while being lifted. In these subfigures, it can be seen that the proposed topic-based `ros2_control` setup does not experience the same periodic noise as the original service-based setup. It can also be seen that the spread of the noise of the original service-based setup at 10 Hz is significantly less compared to the same setup at 20 Hz (as can be seen in Fig. 9). This is caused by the frequency higher frequency not being a divisor of the `telemetrix` frequency of 50 Hz, causing every other control loop iteration to have an extra message. The original implementation does not adjust for these extra messages as described earlier.



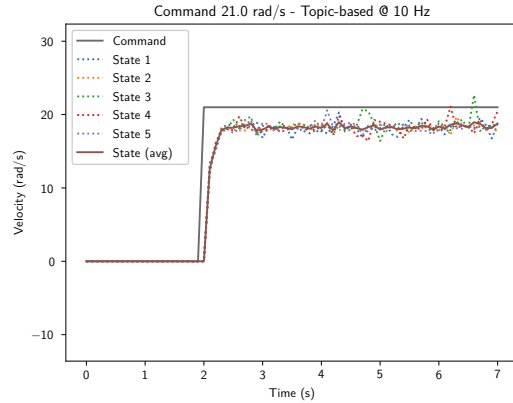
(c) Step response of the original service-based hardware interfaces at 10 Hz with a step input of 4.5 rad s^{-1} while being lifted



(d) Step response of the proposed topic-based hardware interfaces at 10 Hz with a step input of 4.5 rad s^{-1} while being lifted



(e) Step response of the original service-based hardware interfaces at 10 Hz with a step input of 21.0 rad s^{-1} while being lifted

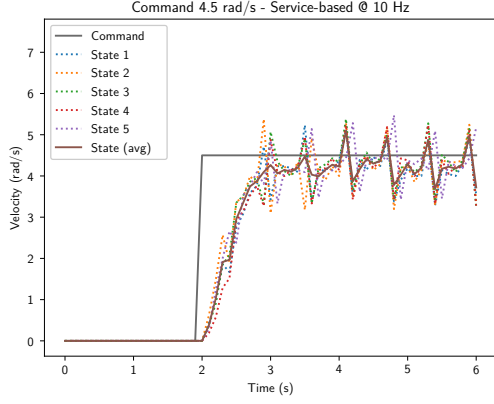


(f) Step response of the proposed topic-based hardware interfaces at 10 Hz with a step input of 21.0 rad s^{-1} while being lifted

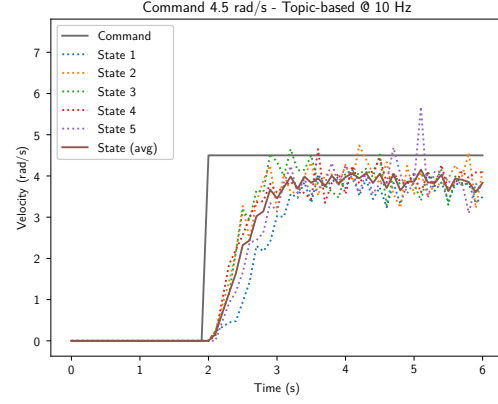
Figure 16: Step response of the `ros2_control` hardware interfaces at 10 Hz with step inputs of 3.0 rad s^{-1} , 4.5 rad s^{-1} and 21.0 rad s^{-1} while being lifted. In these subfigures, it can be seen that the proposed topic-based `ros2_control` setup does not experience the same periodic noise as the original service-based setup. It can also be seen that the spread of the noise of the original service-based setup at 10 Hz is significantly less compared to the same setup at 20 Hz (as can be seen in Fig. 9). This is caused by the frequency higher frequency not being a divisor of the `telemetrix` frequency of 50 Hz, causing every other control loop iteration to have an extra message. The original implementation does not adjust for these extra messages as described earlier. (cont.).

B.4 Step Response - Grounded

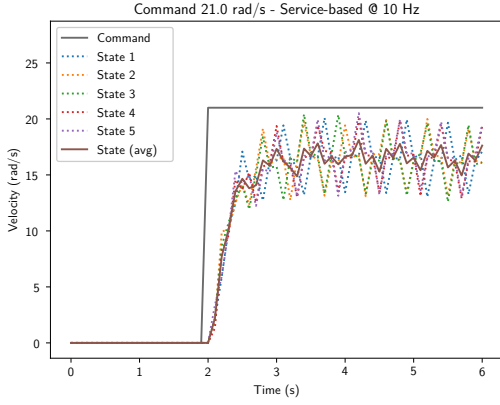
This section contains Fig. 17, which is the step response graphs measured while being grounded for the front left wheel using the original and proposed `ros2_control` setups running at 10 Hz.



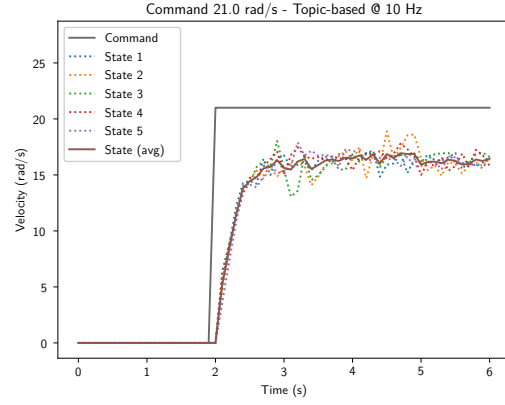
(a) Step response of the original service-based hardware interfaces at 10 Hz with a step input of 4.5 rad s^{-1} while being on the ground



(b) Step response of the proposed topic-based hardware interfaces at 10 Hz with a step input of 4.5 rad s^{-1} while being on the ground



(c) Step response of the original service-based hardware interfaces at 10 Hz with a step input of 21.0 rad s^{-1} while being on the ground

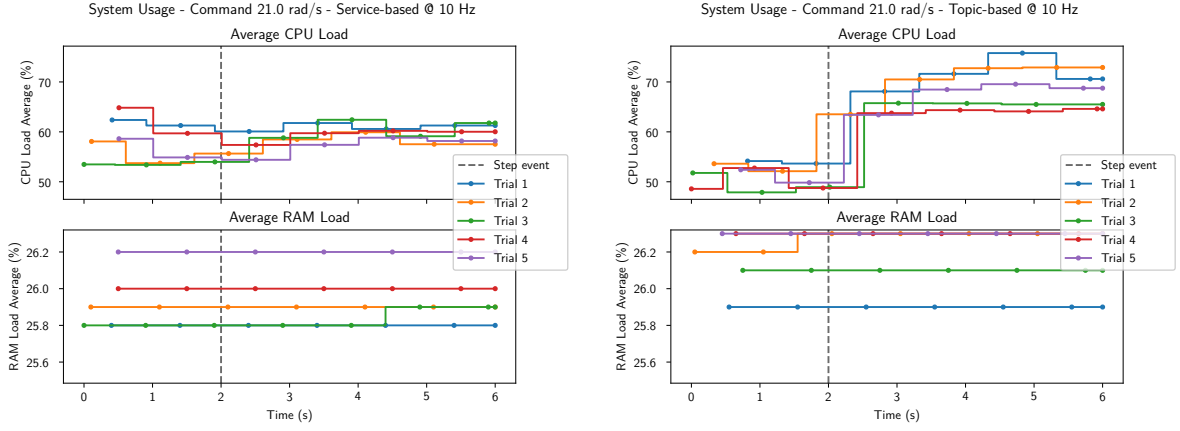


(d) Step response of the proposed topic-based hardware interfaces at 10 Hz with a step input of 21.0 rad s^{-1} while being on the ground

Figure 17: Step response of the `ros2_control` hardware interfaces at 20 Hz with step inputs of 4.5 rad s^{-1} and 21.0 rad s^{-1} while being on the ground. It can be seen that, similarly as in the 20 Hz configurations (Fig. 10), the proposed topic-based setup has significantly less noise than the original service-based setup, and that the original hardware interface reports periodic velocity spikes due to its velocity calculation method. Furthermore, it can be seen that the additional dynamics of the robot dominate the wheel velocity system response, which can be experienced by the increased rise time.

B.5 Step Response - System Resources

This section contains Fig. 18, which is the system resource utilization graph during step response measured while being grounded using the original and proposed `ros2_control` setups running at 10 Hz.



(a) System resource usage of the MIRTE Master using the original service-based hardware interfaces at 10 Hz during a step response with a step input of 21.0 rad s^{-1} while being on the ground

(b) System resource usage of the MIRTE Master using the proposed topic-based hardware interfaces at 10 Hz during a step response with a step input of 21.0 rad s^{-1} while being on the ground

Figure 18: System resource usage of the MIRTE Master during the step response measurements of the `ros2_control` hardware interfaces at 10 Hz excited with a step input of 21.0 rad s^{-1} while being on the ground. It can be seen that the proposed topic-based setup has a lower CPU utilization when the robot is idle compared to the original service-based setup. It can also be seen that the RAM utilization is not impacted significantly.

C Code & Data Availability

In this appendix, the data and software results will be shared. The recorded data and the additional figures can be accessed at <https://jvbrakel.stackstorage.com/s/master-thesis-data>. The software repositories involved in the generation and processing of this data are referenced below, accompanied by a short explanation of their contents.

mirte-thesis-control This repository is specific to this thesis, as it contains the ROS packages related to measuring and processing the system responses. It includes the following packages:

wheel_response_controller This ROS package contains the custom `ros2_control` controllers used to drive the command signals for the step and frequency response measurements.

wheel_response_recorder This ROS package contains the measurement direction node, which is used to manage the setup and recording of each trial. This package also contains all recording launch-files and configurations.

thesis_data_processing This package contains all logic related to the processing of the data recorded with the `wheel_response_recorder` package. It is responsible for data extraction and figure generation.

mirte-ros-packages This repository contains all the pre-existing MIRTE specific ROS packages and the proposed `mirte_modular_hardware` and `mirte_master_control` packages. These proposed packages have been added in addition to necessary patches for the pre-existing packages to support ROS 2 Jazzy. Additional work has been done to enhance this work, which includes specific `telemetrix` patches to add features such as the multi-motor velocity control topic, the update-only encoder topics, and the toggleable service introspection support. Furthermore, a number of more general patches have been done to simplify this work, such as environment variable controlled switching of the `ros2_control` setup.

mirte-sd-image-tools This repository contains the software image building setup, which needed to be patched to build images based of Ubuntu Noble (24.04) instead of Ubuntu Jammy (22.04). This was mandatory as ROS 2 Jazzy targets Ubuntu Noble. Initial work was also done to allow for the creation of ROS 2 Jazzy images using the real-time capable `PREEMPT_RT` Linux kernel, which was later determined to be impractical for general MIRTE deployment.

mirte-install-scripts This repository contains the install scripts used in the creation of a MIRTE software image, which needed to patch target ROS 2 Jazzy and Ubuntu Noble (24.04) instead of ROS 2 Humble and Ubuntu Jammy (22.04). This is a non-trivial task as changes have been made between these two Ubuntu versions with regards to system-wide Python package installations [37], among others.

Telemetrix4RpiPico The used version of the `telemetrix` firmware for the Raspberry Pi Pico. It contains the initial structure for clocksync between the microcontroller and the SBC, however this has not been explored further as the work required to make this work properly with the feature detection system was outside the scope of this work.

ros2_astra_camera Some minor patches to the Astra depth camera driver were required in order to make it build on ROS 2 Jazzy. These were compatible with the ROS 2 Humble and have therefore already been merged.

launch_ros2_control This is a custom `launch` extension for the `ros2_control` to simplify the spawning controllers with dynamic configurations. This `launch` extension is planned to become part of `ros2_control` as discussed in [ros-controls/ros2_control#2454](#).

C.1 Collateral Contributions

This work has resulted in several contributions to core ROS-related projects. This is important as maintaining custom patched versions of libraries for the MIRTE project limits compatibility with the ROS ecosystem while increasing the development workload significantly. The most important contributions are summarized in this sub-appendix.

Lifecycle Transition Event Timestamping

As mentioned in Section 5, the `timestamp` field of `lifecycle_msgs/msg/TransitionEvent` messages did not get populated when such a message was published in any ROS 2 distributions, such as ROS

2 Jazzy ([ros2/rcl/#1019](#), [ros2/rclcpp#564](#)). Furthermore, this field was of type `uint64` instead of `builtin_msgs/msg/Time`, which is more conventional for timestamps ([ros2/rcl_interfaces#184](#)). These issues have been resolved for ROS 2 Rolling and future ROS 2 distributions by upstreamed patches ([ros2/rcl_interfaces#185](#), [ros2/rcl#1269](#), [ros2/rclcpp#2967](#), [ros2/rclpy#1528](#), [ros2/rcl#435](#), [ros2/launch_ros#495](#)).

ros2_control Contributions

Several contributions to the `ros2_control` ecosystem have been made as a result of this thesis, as it uses `ros2_control` extensively. Two notable contributions will be highlighted here.

An issue with the `DiffDriveController` has been identified during the initial testing of the proposed hardware interfaces¹⁰, where a feedback signal was required to be present when the `open_loop` parameter was enabled ([ros-controls/ros2_controllers#1729](#)). Furthermore, a regression in the controller spawner node was identified, which prevented the controller from being spawned correctly ([ros-controls/ros2_control#2385](#)). Both issues were patched and back-ported to the relevant ROS distributions ([ros-controls/ros2_controllers#1730](#), [ros-controls/ros2_control#2386](#)).

REP improvements

Open Robotics has decided to update the ROS Enhancement Proposal (REP) procedures and website. It was crucial that the historical REPs remain accessible, as these contain standards and specifications, for example, REP-107 [23] which specifies ROS' diagnostics system. Therefore, an effort had been made to migrate the existing REPs to the new system. This was partially automated, which broke the content of several REPs ([openrobotics/reps#10](#), [openrobotics/reps#17](#), [openrobotics/reps#18](#)). All of these issues have been addressed and corrected appropriately ([openrobotics/reps#11](#), [openrobotics/reps#25](#), [openrobotics/reps#26](#)).

xacro Boolean Conversion

During this thesis, an inconsistency in the exposed functions of `xacro` was identified where the `bool` was documented as exposed, similar to the other type conversion methods, while this was not the case ([ros/xacro#369](#)). This minor inconsistency has been patched by exposing the `bool` function and has been merged to the upstream repository ([ros/xacro/#371](#)).

¹⁰The proposed hardware interfaces were applied to the MIRTE Pioneer during initial testing, due to the simplicity of the MIRTE Pioneer's `ros2_control` setup.