Precision Robotics with Mid-Weight Hardware and Software: A Swerve Drive Implementation

Robert Henry Forsyth Jr

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of
Master of Science
In
Computer Engineering

Dr. Thidapat Chantem, *Chair*
Dr. Ryan Williams
Dr. Scott Ransbottom

May 12th, 2025
Blacksburg, VA

Keywords: Robotics, Swerve Drive, Intel RealSense D435i, slam_toolbox, Nav2, PID tuning, Gazebo, twist_mux, ROS2, simulation, path planning, cost map, pose, quaternion, navigation stack, depth camera, four-wheel drive, omnidirectional movement, software stack, hardware architecture, general purpose robotics, computational power, optimization, real-world application, robotic design

Precision Robotics with Lightweight Hardware and Software: A Swerve Drive Implementation

Robert Henry Forsyth Jr

ABSTRACT

The era of AI demands ever increasing computational power. As we dawn the era of general-purpose robotics, this trend continues. For complex robotics systems, thoughtful choices of both software and hardware architecture are required for maximizing performance: how do we deploy the most complex software systems on the most minimal hardware all while maximizing usability and reducing error? This question impacts the safety of future systems, as robotic error can mean anything on the order of property damage to human endangerment. This paper explores an advanced robotic system with lightweight compute hardware: a swerve drive robot, leveraging an Intel RealSense D435i depth camera, to navigate a room using the slam toolbox and Nav2 stack. This robot uses a Raspberry Pi 5 as its main compute. A swerve drive is a unique hybrid of traditional four-wheel drive and mecanum drive systems. Pivoting radially, they offer the ability to strafe and the ability to turn in place. This design, built with eight motors, is intended for factories and industry where moving high-value, high-weight products in a controlled manor is critical. Using Gazebo as the simulation environment, and with the eventual goal of moving the software stack to a physical robot, this paper explores the development pipeline for robot, and how to leverage computationally light algorithms for a high-yield product. It also explores considerations for compute utilization to prevent system bottlenecks. The overall goal is to provide a roadmap for researchers and developers who are interested in building and optimizing swerve drive robots using ROS2 on less expensive hardware, specifically the Raspberry Pi 5.

Precision Robotics with Lightweight Hardware and Software: A Swerve Drive Implementation

Robert Henry Forsyth Jr

GENERAL AUDIENCE ABSTRACT

This paper discusses the development and optimization of a four-wheel robotic system. It highlights the importance of selecting appropriate software and hardware for achieving high software complexity and low error on a low power compute. The study involves a robot equipped with a depth camera for navigation, utilizing specific software tools for mapping and control. The system is tested in various environments, both in simulation and with the aim of real-world application.

DEDICATION

I would like to thank my committee members, Dr. Thidapat Chantem, Dr. Ryan Williams, and Dr. Scot Ransbottom, for their guidance and support throughout this project. I would also like to thank David Spadaccia, Michael Yanoshak, and Catherine Hebert for encouraging me through this process. Finally, I would like to thank my family for their support.

KEY TERMS

"Stack", "Software Stack", "Software Suite": A collection of software tools that work together to achieve a common goal. Typically used with more nuiance, but for the purpose of this paper they will mean essentially the same.
PID Tuning: A method of adjusting the parameters of a proportional-integral-derivative controller to achieve optimal performance.
Gazebo: A robot simulation environment that allows for testing of robotic systems in a virtual environment.
slam_toolbox: A set of tools for performing simultaneous localization and mapping (SLAM) in robotic systems, specifically ROS.
Nav2: A navigation stack for ROS2 that provides tools for robot navigation.
Swerve Drive: A type of robotic drive system that allows for omnidirectional movement.
Cost Map: A map of the environment that assigns a cost to each cell based on its traceability. Obstacles are assigned a higher cost, while open areas are assigned a lower cost. Used for path planning.
Pose: The position and orientation of a robot in space. This is typically represented as a 3D coordinate and a quaternion.
Quaternion: A mathematical construct used to represent rotations in 3D space. It is a more compact representation than Euler angles. Instead of using a six number representation, it uses four numbers to represent the same information. This approach is more efficient for computation and avoids issues with gimbal lock.
Swerve Module: The individual wheel assembly of a swerve drive robot. Each module can pivot independently to allow for omnidirectional movement.
Swerve Drive Radial Axis: The axis around which the swerve module pivots. This is typically perpendicular to the ground and parallel to the wheel. Used synonymously with "pivot axis".

Swerve Drive Axial Axis: The axis along which the wheel rotates. This is typically parallel to the ground and perpendicular to the swerve module. Used synonymously with "drive axis".
Twist_mux: A tool used to manage multiple sources of velocity commands in ROS2.
"Differential Drivetrain" vs "Swerve Drive Differential": A differential drivetrain is a type of drive system that uses two wheels to control the direction of the robot. A swerve drive differential describes the differential drive system of a swerve drive robot. This entails two motors on a module, which depending on the combination of commands to the two motors and their difference in motion can yield either a radial, axial, or radial and axial motion. This is a unique feature of swerve drive robots, and is not present in traditional differential drive systems.
MVP or Minimum Viable Product: The simplest version of a product that can be released to the market.
Puck Lidar: A type of Lidar sensor that is used for navigation and mapping in robotic systems. It is a compact and lightweight sensor that is ideal for use in small robots. It's field of view is 360 degrees, with most having a range of 10-20 meters. It is typically used for indoor navigation and mapping.
IMU or Inertial Measurement Unit: A sensor that measures the orientation, velocity, and gravitational forces acting on an object. It typically consists of an accelerometer, gyroscope, and magnetometer. It is used for navigation and control in robotic systems, usually alongside other sensors to prevent IMU drift (error accumulation over time due to the antiderivative of sensor data).
SLAM: Simultaneous Localization and Mapping. A technique used in robotics to create a map of an unknown environment while simultaneously localizing the robot within that environment. Aids in generating a proper transform between the robot and the world frame (typically the base_link and map frame in ROS)
TF: The ROS ecosystem transformation tree topic which allows physical bodies to register their position based on a parent transformation frame. TF2 is its successor, which is what is used in this paper. However, as shorthand, *tf* within this paper refers to tf2.
URDF: Universal Robot Description Format. A file format used to describe the physical properties of a robot, including its geometry, kinematics, and dynamics. It is used in ROS to represent the robot model and is typically used in conjunction with Gazebo for simulation.
SDF or Simulation Description Format: A file format used to describe the physical properties of a robot and its environment. It is used in Gazebo for simulation and is typically used in conjunction with URDF for representing the robot model.
ROS2 or Robot Operating System 2: An open-source framework for building robotic systems. It provides a set of tools and libraries for developing robot software and is widely used in the robotics community.
Bench Test: A test performed on a robot or robotic system to evaluate its performance and functionality. A "Bench Test" usually happens on a "Test Bench", which is a physical or virtual environment where the robot can be tested and evaluated. This is typically done before deploying the robot in a real-world environment.

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

Swerve drive robots are a unique hybrid of traditional four-wheel drive and mecanum drive systems. With wheels that can pivot radially, they offer the robot the ability to strafe as well as turn in place. These robotic systems are commonplace within the First Robotics Competition (FRC) community [13], where they are used to navigate complex obstacle courses. This work has, however, not been a large interest of the ROS2 collegate research community, where these robots are rare. This paper aims to bridge that gap by providing a detailed overview of the development and optimization of a swerve drive robot within the ROS2 ecosystem. The research also details optimization of perception methods for lightweight compute hardware, specifically as it pertains to navigation and mapping. The goal is to provide a roadmap for researchers and developers who are interested in building and optimizing swerve drive robots using ROS2 on less expensive hardware, specifically the Raspberry Pi 5.

Robots are becoming increasingly important in a variety of fields, spanning applications from military to food delivery. As robots become more integrated into our daily lives, they must perform more complex tasks and operate in more challenging environments. One area of research that has gained significant traction in recent years is the development of autonomous robots that can navigate and map their environment, particularly leveraging SLAM to facilitate this. Reliable methods of this system are critical for areas such as search and rescue, where robots need to be able to navigate through unknown and potentially hazardous environments. It is also important for factories, where supplies need to be moved efficiently and accurately. To succeed in industry, robots must achieve these objectives while using minimized hardware to increase margins. The development of swerve drive robots that use off-the-shelf hardware is an important step towards achieving this goal, as they offer a unique combination of mobility and maneuverability that is well-suited for navigating complex environments. While they come with their own set of challenges, namely their energy consumption and mechanical complexity compared to standard differential drive robots, they are a promising solution for a variety of industrial applications.

Current research in this field has used highly simplified and, as a consequence, limiting robotic designs for swerve drive robots. Many rely on slip rings, or motors mounted directly onto the wheels. While there are examples such as NASA's swerve drive car that demand motors connected directly to the wheels for added power, there are better design decisions that can be made mechanically to alleviate the electrical complexities. Instead, we can use a swerve differential drive module, where two pulleys work in tandem to both drive the wheel, and drive the rotation of the wheel module. This yields a more compact design and reduces the number of moving electrical components.

The introduction of SLAM algorithms, working in conjunction with NAV2, also presents a unique opportunity to marry the best hardware and software solutions to create a robot that can navigate and map its environment in real-time. Deployed to a Raspberry Pi 5, the goal is to deploy this sophisticated software stack on lightweight hardware, keeping a close eye on the hardware demands of the algorithms. The research will also strive to keep track of robot error, measuring offset from its desired goal and overshoot of the modules to optimize the PID

parameters for robotic efficiency. This paper presents a comprehensive overview of the development and optimization of swerve drive robots within the ROS2 ecosystem. It focuses on strategies to reduce movement error and minimize computational load, particularly with lightweight hardware that does not rely on extensive parallel processing. Additionally, it aims to serve as a practical roadmap for researchers and developers interested in building and refining their own swerve drive robots using ROS2 and Gazebo.]

Additionally, a note should be made about the "mid-weight" qualifier I provide within the title of this paper. Across the spectrum of devices, I would consider a Raspberry Pi middle to low weight. Any lower and you would find yourself using hardware like an Arduino or Feather, which fall within the MCU environment. Any higher, and you would leverage a standard PC or, on the higher end, anything with a GPU (NVIDIA Jetson, Orin, or a desktop PC with the like). Super-weight would be a server cluster, quantum computer, or any industrial applications of a distributed compute system. This is the gradient within the mind of the researcher and, as such, is why I will use the terms "mid-weight" or "lightweight" to describe the Raspberry Pi 5.

PROBLEM STATEMENT

The problem this paper aims to address is the lack of information on the development and optimization of swerve drive robots within the ROS2 ecosystem. While swerve drive robots are common in the FRC community, they are rare in the research community. This paper aims to bridge that gap by providing a detailed overview of the development and optimization of this ROS2 robot, providing a roadmap for researchers and developers who are interested in building and optimizing swerve drive robots using ROS2. This paper also strives to use non-GPU enabled hardware systems, but write the software in such a way to prepare for the AI/digital twin era that the NVIDIA Jetson and others are beginning to usher in. This goal hopes to solve the migration problem for when projects start adopting higher-power solutions.

RESEARCH QUESTIONS

- How can we use ROS2 and Gazebo to simulate and validate our swerve drive robot?
- How can we configure Nav2 for the most computational headroom without increasing error?
- What perception systems are best suited for swerve drive robots? What are the trade-offs between different perception systems?
- If it's measurable it's manageable. What are the key performance metrics for swerve drive robots?
- What steps can we take to make data aggregation easier as we move to a digital twin infrastructure?

RESEARCH OBJECTIVES

- Develop a Swerve Drive robot in Solidworks, accounting for proper frames of reference for the kinematic model of the robot
- Export the URDF model to Gazebo and Rviz for simulation
- Implement the slam_toolbox and Nav2 stack for mapping and navigation

- Test the robot in a variety of environments within simulation, and prepare the software architecture for deployment on a physical robot
- Configure the software stack for the Raspberry Pi 5, ensuring CPU usage is kept below 70% and RAM usage is kept below 70% under normal operating conditions
- Publish key topics to the ROS network that can be used for data capture later for AI applications

SCOPE AND LIMITATIONS

The scope of this paper is on traditional methods of robot development as they relate to ROS2. While this paper will branch into leveraging machine learning as a means of optimizing the robot's PID parameters, as well as discussions of A* search patterns for navigation, software like Issac Lab or Omniverse are not within the scope of this paper. This paper will make suggestions at times on how to leverage produced features for use with AI in the future. These will not be overly detailed as that would detract from the focus of the paper, but these are areas of interest of the author, and will be explored in future works.

PRELIMINARIES

To provide insight into technologies discussed within this paper, the following is provided for convenience:

ROS2

ROS2 is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a variety of robotic platforms. One misconception is that ROS2 is real-time. It is not but provides the flexibility for developers to implement real-time systems. The foundations for this project relies on two models of the ROS2 ecosystem: the Publisher-Subscriber model and the Client-Server model. The former is used for communication between nodes, while the latter is used for within parts of the Nav2 stack. This study presents several diagrams of the ROS2 architecture, to provide a clear understanding of the system. For this research, it is critical to understand the decision to use ROS2. More than just being the state of the art, its communication protocol (DDS) is more fault tolerant than the ROS (ROS1) protocol, which leveraged UDP and TCP. On the surface there is no problem with applying these protocols, but a single ROS master node to coordinate all the nodes in the system is a single point of failure. This is not the case with ROS2, which is a distributed system. This will be discussed briefly within the context of the paper, but the reader should be aware of this distinction. Note that, when a distinction needs to be made between ROS1 (ROS) and ROS2, this paper favors explicitly calling it "ROS1" to provide clarity, despite this not being the standard convention. Using the term "ROS" will indicate a generalization of the two, where either can apply.

slam_toolbox

The slam_toolbox is a set of tools for performing simultaneous localization and mapping (SLAM) in robotic systems. It is a collection of ROS2 packages that help generate a 2D map of

the robot's environment and localize the robot within that map. The slam_toolbox is used in this study to generate a map of the robot's environment and localize the robot within that map. The slam_toolbox is a key component of the Nav2 stack, which is used for robot navigation. In terms of this study, the slam_toolbox exclusively focuses on mapping, which the Nav2 stack leverages to generate a "cost map" of the environment, which is used for path planning.

Nav2

Nav2 is a software stack for ROS2 that provides tools for robot navigation. It is a collection of ROS2 packages that help the robot navigate its environment by generating a "cost map" of the environment and planning a path through that map. The "cost map" can be thought of as a gradient, where obstacles take on a darker hue, with higher costs. To stick with this mental image, that means our traversal will remain in the lighter, lower-cost troughs of the cost map. Nav2 is supported by the Open Navigation Stack (OpenNav) project, which aims to provide a flexible and extensible navigation stack for ROS2. In the case of this study, we use the "Waypoint" modality, where the robot localizes and estimates its starting pose and is then commanded to achieve a different pose.

Gazebo and Rviz

Gazebo is a robot simulation environment and physics engine that acts as a "stand in" for the robot and its environment. It is used to test robotic systems in a virtual environment before deploying them on a physical robot. Rviz is a 3D visualization tool for ROS that allows you to visualize the robot's environment and sensor data in real-time. Both tools are used to test the robot's navigation and control algorithms in a simulated environment before deploying them on a physical robot.

LIT REVIEW

INTRODUCTION

The literature on swerve drive systems and navigation in ROS2 is limited, as ROS2 was only introduced eight years ago at the time of this research. As a result, the reviewed papers are largely drawn from the past decade. These papers cover various topics, including fuzzy logic control, motor construction, PID tuning, and the use of different sensors for navigation. The focus of these papers spans mechanical, electrical, and software aspects specific to their proposed solutions. To keep this review concise, key assertions from these papers will be summarized and critiqued, with comparisons and connections being drawn to the research presented in this thesis. This section aims to bridge the gap between the researchers' needs and the objectives of this study. Direct implementations done within the body of this work will be mentioned but will not be discussed directly to avoid confusion. Those implementations will be discussed within the *Results* or *Methods* section of this thesis.

MOTORS AND DRIVE AXIS

There are two main approaches to driving the wheels of a swerve drive system: attaching the motor directly to the wheel or using a belt system [2][3][4][5][6][7]. The former is common in existing literature, but it can lead to wiring issues and increased complexity in the design. In contrast, the research presented in this paper utilizes a belt and pulley system to drive the wheels, which simplifies the design and mitigates potential wiring problems. However, this approach is not without its drawbacks, as it can lead to belt slip and other mechanical issues. Commercially available swerve drive modules, such as those from Swerve Drive Specialties [12], utilize a differential drive system too. The characteristics of the two different approaches are shown below:

Design Choice	Advantages	Disadvantages
Motor on Drive Axis	- Reduced mechanical complexity	- Forbids continuous pivoting due to wire wrap (unless a slip right it used) - Driving wheel directly on motor without proper consideration can yield a bent motor shaft - Increased wiring complexity to account for the drive motor
Swerve Differential	- Reduced wiring complexity - Allows for continuous pivoting	- Increased mechanical complexity - Potential increase in failure rates and maintenance requirements due to increased complexity - Increased computational complexity to account for ratio of motors to wheels

MOTORS AND RADIAL AXIS

Both research and professional swerve drive modules agree that added complexity is not needed for the radial axis. Typically, the radial axis moves the primary axial module with either belts, or gears. The research presented in this paper utilizes a belt system to drive the wheels, which simplifies the design and mitigates potential wiring problems. To account for belt slip, the research adds a homing limit switch. In other conditions, you could satisfy this with a gear ratio attached to an absolute encoder, but since absolute encoders are expensive, this is a good compromise. Additionally, integrating this system would increase the physical size of the proposed module, which is not ideal for simplicity. There are designs that chain the radial wheels together but this contravenes the idea of a swerve drive system and limits mobility [8].

GEARS VS BELT SYSTEM

The research presented in this paper utilizes a belt system to drive the wheels, which simplifies the design and mitigates potential wiring problems. At high speeds or under force conditions adverse to the commanded motion, the belt system can slip. This is not a problem with gears, but gears that meet the size requirement of this outdoor swerve drive spec are unavailable or monetarily prohibitive. The research presented in this paper utilizes a belt system to drive the wheels, which simplifies the design.

WHEEL TUNING

There are three schools of thought for tuning the motors of a swerve drive system: PID tuning, fuzzy logic, or reinforcement learning to tune the control based on a variety of environmental factors. The research presented in this paper utilizes PID tuning, as it is the most common and well-understood method for tuning motors in a swerve drive system. Fuzzy logic is less common, revolving around discrete controls given an input speed [1]. However, this usecase can be covered through PID tuning, and even better through a discretized PID tuning algorithm (shown later). Reinforcement learning is an emerging field that has the potential to revolutionize the way we tune motors in a swerve drive system, and is quickly becoming common practice. A great example of this is the use of Issac Sim to train Blue [22], a two-legged Star Wars robot to walk. This is outside the scope of this research, but it is an interesting area of study.

CONTROL SYSTEMS

The research presented in this paper utilizes a web app to control the robot, which is a novel approach that is not commonly used in the related literature. Most research uses a joystick or other physical controller to control their robot(s) [7]. The web app allows for greater flexibility and ease of use, along with better visualization of the solution set, as it can be accessed from any device with a web browser. This is a significant advantage over traditional controllers, which are often limited to a single device. The web app also allows for greater customization and control over the robot's behavior, as it can be easily modified to suit the user's needs, allowing it to grow with research need. This is a similar approach to that taken by Foxglove [25], an RVIZ alternative that not is confined to single-computer deployment. Additionally, instead of directly controlling the robot, the controller is piped into the control logic through a twist_mux, a

multiplexer that uses priority to determine which command to execute. Without this, there would not be a convenient way to switch between human control and autonomous control, beyond a mode handler (common in industry, with an industrial application being the Daimler AP4 chassis) [26].

3D PRINTING FOR CHASSIS

This research leans into the 3D printing of parts to reduce the cost of the robot. This is becoming more common in literature, but does not often appear in conversation when the problem set is software driven. However, those who care about software make their own hardware, and thus it was important to develop a reliable mechanical system that would interface cleanly with the software stack. This is a significant advantage over traditional methods, which often rely on expensive and complex hardware. The 3D printed parts used in this research are not only cost-effective, but they are also lightweight and easy to manufacture.

TURTLEBOT PARALLELS

The "Turtlebot" [14] is a small, affordable robot commonly used in research and education. The latest version, Turtlebot4, uses both a puck lidar and a depth camera for perception. Similarly, this research uses a depth camera and aims to provide a basic framework for building a controller. The robot design presented is a minimal viable product (MVP) of a swerve drive robot, meant as a starting point—not a complete solution—for further development and research, especially in tuning and building such systems.

SLAM TOOLBOX vs VOXEL MAPPING

While the Nav2 stack supports voxel grids out of the box [17], the presumed additional processing used to determine and maintain this data with online learning was contrary to the hardware selection and goal of the research, which is intended to be minimal (Raspberry Pi 5). Instead, we relied on a basic perception model which we could tune for the environment to capture the map similar to [9][11], passing it to SLAM toolbox [16], and then supply it to Nav2[15]

METHODOLOGY

INTRODUCTION

This section details the methods and results of the developed swerve drive robot in ROS2, including the development of the robot model, the implementation of the slam_toolbox and Nav2 stack, and the testing of the robot in simulation. The section is divided into several subsections, each detailing a specific aspect of the development process. The process unfolds chronologically as the researcher experienced them. Note that all source code and models are available on the GitHub repository for this project; the link is provided in the appendix. As this work is ongoing, attached to the GitHub link in the appendix is a specific commit hash. This hash corresponds to the time this paper was written and is what you should reference if the repository changes.

ADDRESSING THE RESEARCH QUESTIONS

In this section you will find a chronological breakdown of the research questions, and how they were addressed. To permit a better organization of how the questions are addressed, the following breakout is consistent with the research questions:

How can we use ROS2 and Gazebo to simulate and validate our swerve drive robot?
This is covered in the section "Validating the Swerve Drive Robot in Gazebo"

How can we configure Nav2 for the most computational headroom without increasing error?
This is covered in the section "Validating Deployment on a Raspberry Pi 5: Checking Computational Headroom"

What perception systems are best suited for swerve drive robots? What are the trade-offs between different perception systems?
This is covered in the section "Deploying Nav2 and Slam Toolbox, With Validation"

If its measurable its manageable. What are key performance metrics for swerve drive robots?
This is covered in the section "Developing a Swerve Drive Module Using a Swerve Differential"

DEVELOPING A SWERVE DRIVE MODULE USING A SWERVE DIFFERENTIAL

At the outset of this section, understand that there is a series of robots known as "differential drive" robots. These robots are designed with two wheels, and their motion is derived from a "differential" of the two wheels. This is a common design, and is used in many robots today. The swerve drive module I propose operates using a difference, also known as a differential, between two pulleys, relying on the encoders in the motors and a homing limit switch to determine the position of the wheels. The use of the word "differential" in these two cases is problematic, even though its usage is appropriate. As such, when discussing the swerve drive module, I will refer to it as a "swerve differential" to avoid confusion.

This swerve differential allows you to avoid slip rings for motors and encoders on the drive axis, which simplifies the electrical complexity. Introducing this swerve differential does require the use of bevel gears to modify the orientation of the motor shaft to the wheel shaft. See the diagram below for how this is accomplished:

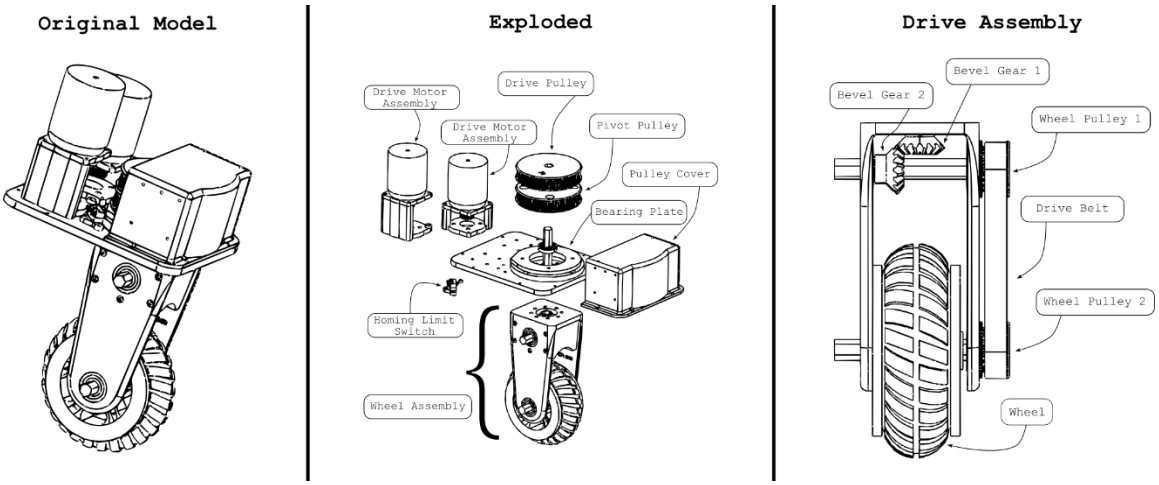


Figure I: The swerve drive module

In developing this module, we now have an intuition for which solid bodies should operate as a link, and which should operate as a joint. This will be critical as we move into the URDF development.

It is also important to note what we are "validating" and what the acceptance criteria are. The acceptance criteria for the swerve drive module is as follows:

- Keep Raspberry Pi 5 computational headroom in mind, below 70% utilization on a full bench test.
- The robot must not deviate from its commanded goal pose beyond 0.1 m
- The robot's control scheme must not move when module error is above 0.1 rads
- The less overshoot in a module, the less resultant error/time lost to convergence. Minimize convergence time and oscillations, preferably below two seconds and any oscillations with an amplitude above 0.1 radian deviation from the goal should be seen as non-compliant.

DEPLOYMENT STRATEGY

The heart of this research revolves around an understanding of both hardware and software, and how to implement a scalable, easy to maintain system. While the former has already been discussed, the deployment strategy for the latter still warrants acknowledgement. ROS is, traditionally, tightly tied to a particular Ubuntu distribution. To simplify the deployment process, the researcher has opted to use Docker containers to deploy the ROS2 system. This allows for a more modular approach, where one container can run all the components. Alternatively, one could deploy each component of the system in its own container. This research uses a single-container deployment approach to reduce multi-container overhead. However, as the totality of the services are broken out into separate launch files, you could run a multi-container deployment with the same built image.

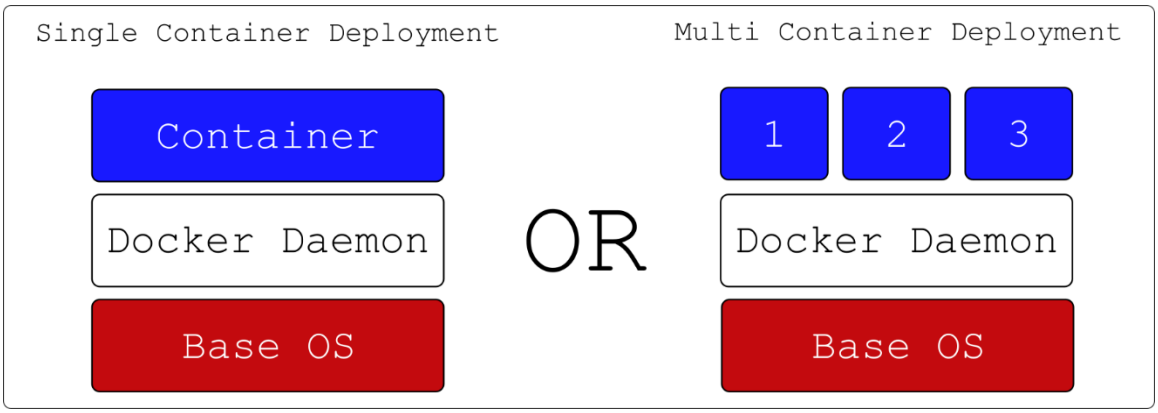


Figure II: Container Implementation

Butler Lampson, one of the first scientists to envision the modern personal computer in 1972 once said that "All problems in computer science can be solved by another level of indirection" [19]. However, it is the belief of this researcher that humans opperate along the inverse of this principle, where many layers of indirection result in confusion. As such, while there are other tools better adapted to generate launch files for robot descriptions (in this case the generated URDF) for specifically ROS2, adding another tooling into the production process is unneccsary. We will instead directly leverage a Solidworks plugin [18] that generates a URDF given that you provide it with which solid bodies make up a link, and which reference frames define that joint's motion. More on the importance of proper link and joint structures will be provided later in the paper. By electing to use the plugin, we keep all our frame definitions and axis in one place, and allows us to leverage the existing tooling for ROS2. The model was developed with the following considerations in mind:

- The frames of the robot must have the following hierarchy, based on Nav2 requirements:
  - base_frame
    - base_link
      - WheelPivotA
        - WheelDriveA
      - WheelPivotB
        - WheelDriveB
      - WheelPivotC
        - WheelDriveC
      - WheelPivotD
        - WheelDriveD
      - camera_pivot
        - camera_link
      - IMU Link
- The X Axis is the most crucial axis to align: for Nav2, **the X axis of the base_link must be aligned with the forward motion of the robot**. This is the default for the URDF exporter, so it is important to ensure that the model is aligned with the X axis in Solidworks.
- For the pivots to function appropriately, the pivot axis must be aligned on their Z axis. Otherwise, you risk improper or lopsided movement of the modules
- The robot must have a means of perception, such that it can navigate the world. This is accomplished using a camera and IMU. The camera is mounted on a pivot, which allows it to

rotate and look in any direction. The IMU is mounted on the base_link and provides data on the robot's orientation and acceleration. It is important to note that the physical bodies of the sensors are available at this stage, but the sensor definitions that Gazebo leverages are added later in the process.

When developing the model, the best approach is to design the model through repeated modules. Following this, you can proceed with adding points and reference axis. Remember, for the sake of simplified axis, choose a common axis convention. For the sake of this paper, our translational axis is always the X axis, whereas rotational is typically the Y or Z axis. This will make it easier to export to URDF, and easier to debug later. Since this is foundational to the structure of the robot, and changing it will require additional work, it is best to get this right the first time. Below is an example a reference frame used in the model. Note the preserved convention:

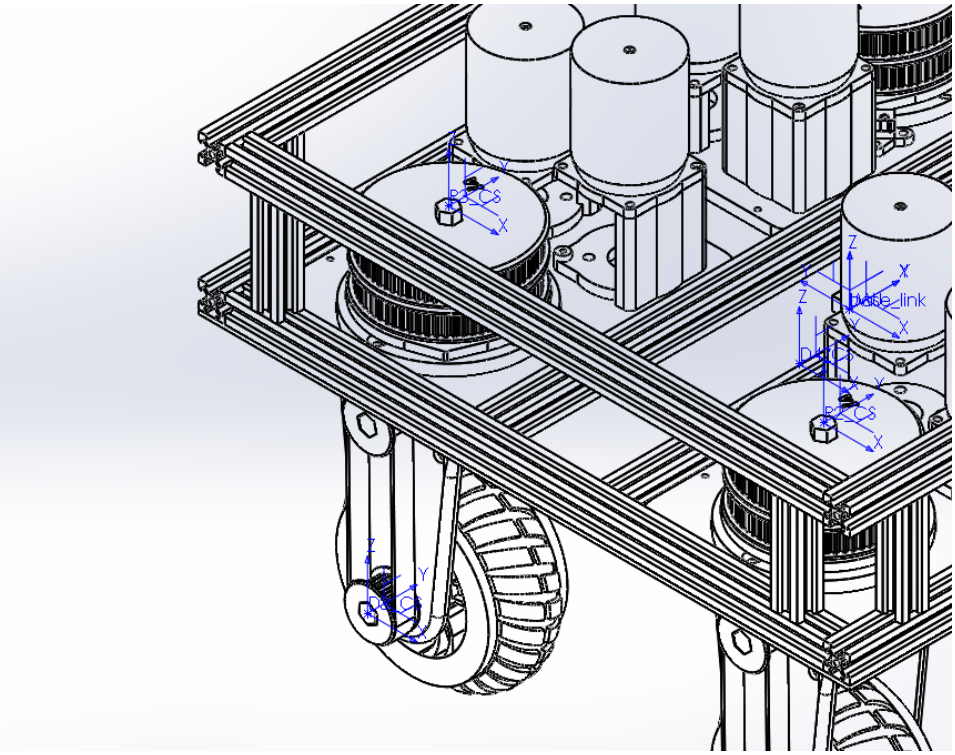


Figure III: URDF Frames Example

Within this example, a point was added where we expected the reference frame to be, then added a reference frame to that point. The Solidworks plugin can do this automatically, but strictly defining your axis was helpful for debugging. Below is a snapshot of the Solidworks Feature Design Tree typically on the left hand side of the software that shows the reference frames and points:

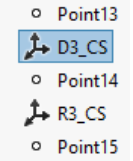


Figure IV: An example of the Solidworks timeline

Once there are defined, you can export the URDF. This is done by selecting the "Export to URDF" option in the Solidworks plugin. This will generate a URDF file, as well as a set of meshes and textures that are used to render the robot in Gazebo. An example of the export plugin is shown below:

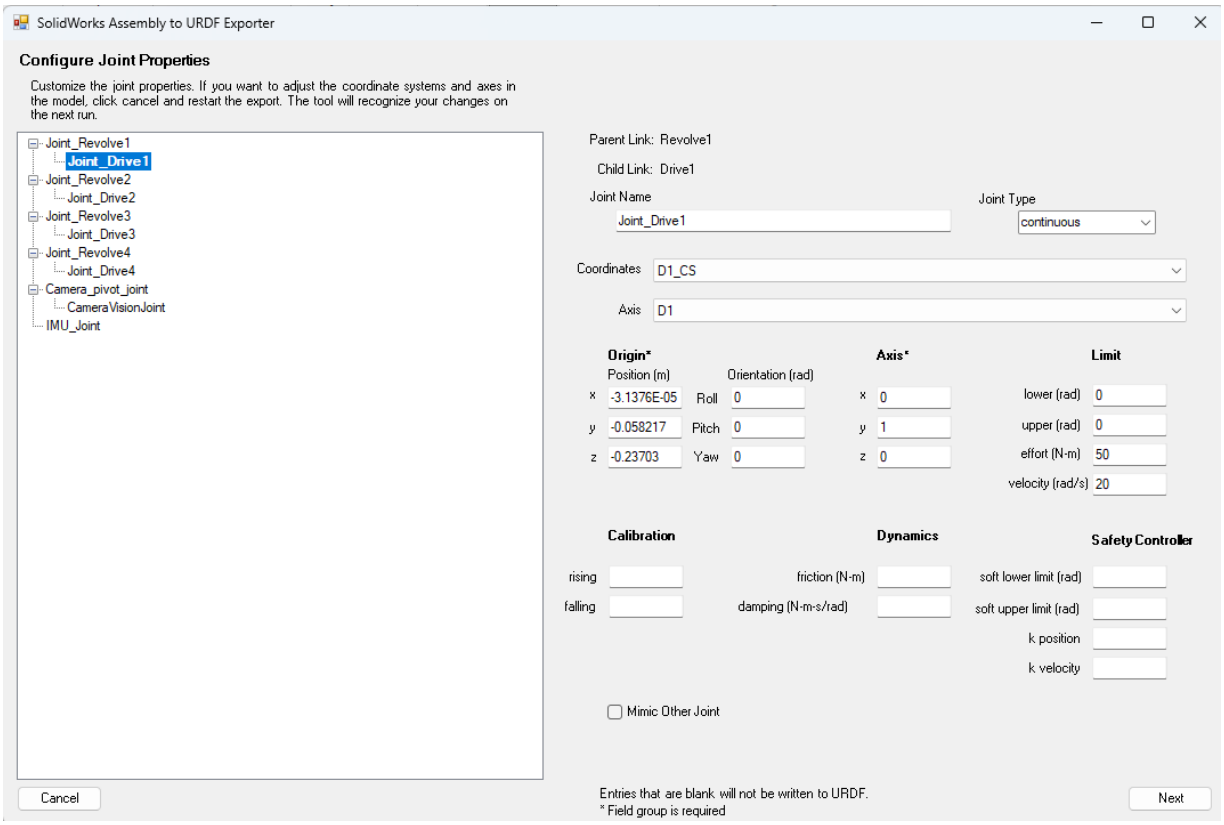


Figure V: An example of the export window

EXPORTING URDF ARTIFACTS

Upon exporting the URDF, you will find the following file structure:

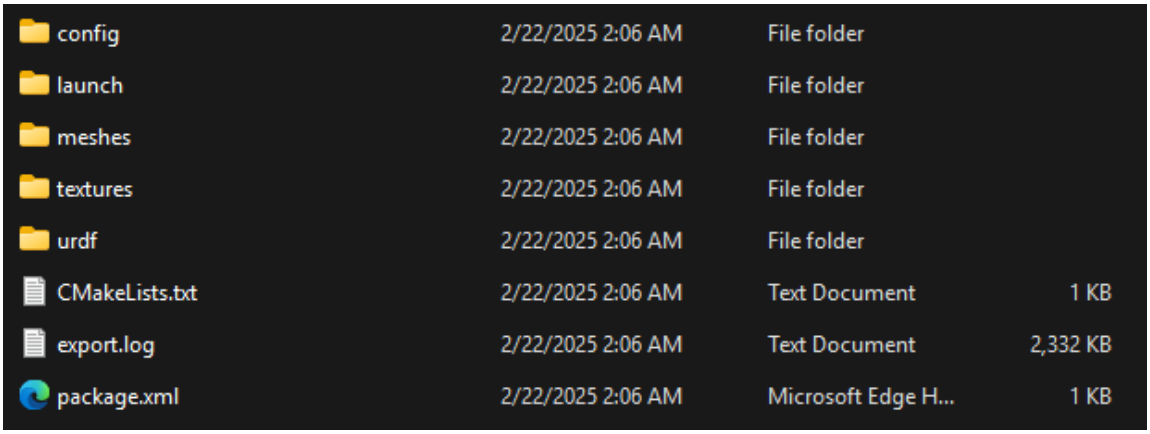


Figure VI: Exported file structure

This structure is consistent with standard description files for ROS1. However, the launch files are not directly compatible with ROS2. This is a current known shortcoming of the export tool. Even so, the work around is simple and will be discussed later in this paper.

When preparing the URDF, there are a few considerations to make. First and foremost, the key framework you are setting up is a link/joint relationship. Links are coordinated through joints, with links typically having some form of mesh or solid that is rendered. This relationship is seen in the urdf segment below:

```
<link
    name="CameraVisionLink">
    <inertial>
      <origin
        xyz="-3.83807569059869E-17 -6.77896212537346E-05 -0.000495383252896442"
        rpy="0 0 0" />
      <mass
        value="0.00122306679751053" />
      <inertia
        ixx="2.45260777746173E-08"
        ixy="1.49835825799552E-22"
        ixz="-7.43449114089117E-22"
        iyy="6.15023324061756E-07"
        iyz="3.26714169017072E-09"
        izz="6.38451387870187E-07" />
    </inertial>
    <visual>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
```

```
          filename="package://hermes_robot_description/meshes/CameraVision.STL" /
>
      </geometry>
      <material
        name="">
        <color
          rgba="0.5 0.5 0.5 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://hermes_robot_description/meshes/CameraVision.STL" /
>
      </geometry>
    </collision>
  </link>
  <!-- Sensors -->
  <gazebo reference="Camera_pivot">
    <sensor name="camera" type="camera">
      <pose>0 0 0 -1.57 1.57 0</pose>
      <topic>image_raw</topic>
      <always_on>true</always_on>
      <update_rate>10</update_rate>
      <visualize>true</visualize>
      <camera>
        <horizontal_fov>1.047</horizontal_fov>
        <image>
          <width>800</width>
          <height>600</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.1</near>
          <far>100</far>
        </clip>
      </camera>
    </sensor>
  </gazebo>
```

Within the urdf model, note that the link and joint, despite being closely related, do not share the same name. This is important, as the URDF will compile incorrectly if the names are the same. The joint and link should operate in a parent/child relationship. Additionally, note that Solidworks has also added the mass and inertia tags. Learned from experience, the Gazebo physics engine is particular about these values being calculated correctly. As was a consequence in simulation, you will find the robot may flip over or behave erratically if you attempt to edit these values manually without considerations being made for the physics engine. Finally, note

the use of a Gazebo tag. This usage is discussed later in the paper, but in concept a URDF is intended to be a physical representation of a robot. However, for the commands to be properly translate between ROS and Gazebo, apart from the actual *Gazebo Bridge*, you need to define a corresponding motion plugin for every critical joint.

ADAPTING THE URDF FOR GAZEBO

Despite the above warning, it should be noted that massless links and plugins do not produce a problem for the physics engine when added into the URDF. For keen eyes, you will notice that there is no native way within this export tool to define the necessary Gazebo plugins to allow for the robot to be controlled. The robot will not be able to be controlled without these plugins, nor will you be able to simulate the sensor data.

As an additional post-processing step, you will need to add your sensors and controllers to the URDF. This is done using the Gazebo tags. The following is an example of the camera sensor:

```
<gazebo reference="CameraVision">
  <sensor name="RealsenseD435" type="rgbd_camera">
    <pose>0 0 0 0 0 0</pose>
    <camera>
      <horizontal_fov>1.25</horizontal_fov>
      <image>
        <width>320</width>
        <height>240</height>
      </image>
      <clip>
        <near>0.3</near>
        <far>100</far>
      </clip>
      <optical_frame_id>CameraVision</optical_frame_id>
    </camera>
    <update_rate>2</update_rate>
    <topic>depth_camera</topic>
    <always_on>true</always_on>
  </sensor>
</gazebo>
```

Finally, once you have prepared this, change the file extension name to .urdf.xacro. This is critical because, on launch, this file is converted to both and URDF and an SDF so that both ROS and Gazebo can consume the proper data for rendering and simulation. Upon starting the system, the URDF file will be published to the /robot_description topic, resulting in the following TF frames:

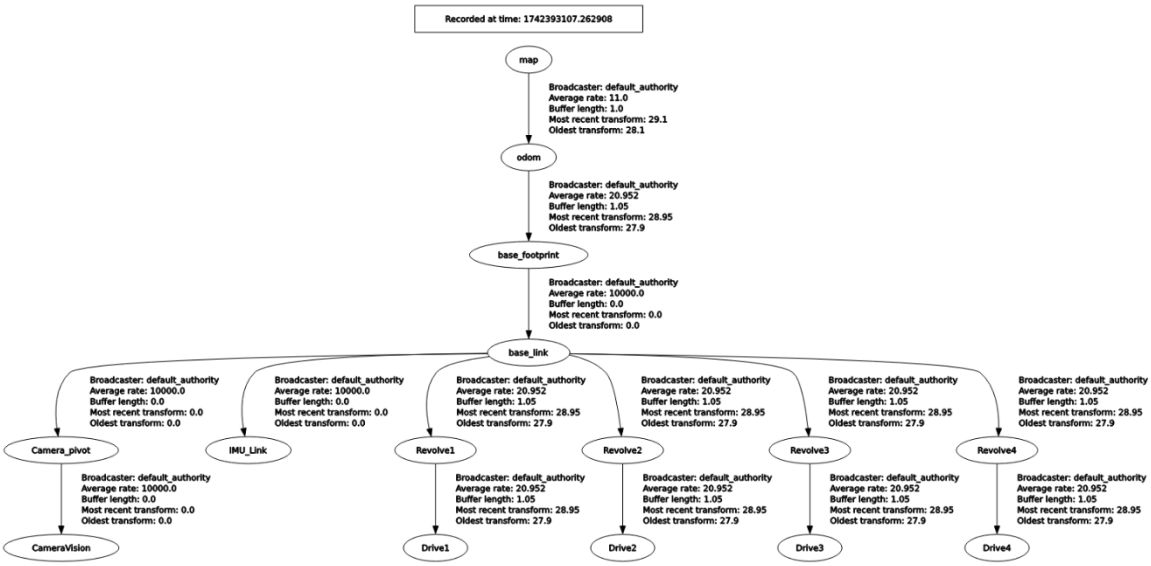


Figure VII: TF Frame representation

WRITING A LAUNCH FILE TO SPAWN URDF

The approach used within this paper deviates from the typical Gazebo workflow by seperating out the world URDF from the robot URDF. In this way, we can leverage one xacro file that can be converted to both SDF and URDF, without having complicate it by wrapping it in world-file-related tags. This is done by adapting the launch file to start Gazebo, generate a world, then spawn in the robot at a given Cartesian coordinate. This also allows you to vary the world specifications, then simply spawn the robot. This paper also utilizes a methodized approach to how the ROS2 launch files are written. Favoring a more modular approach, the typical launch file is broken down into smaller sub-launchers, each with a specific purpose and each written in their own seperate launch file. Upon wanting to bring up a sequence of services, you can either run each sub-launch file individually, or run the main launch file that will bring up all the services in the correct order. This is a more modular approach, and allows for easier debugging and testing of the robot. In accordance with this, this paper has a "world" launch file (world.launch.py) and a "robot" launch file (robot.launch.py). You can also bring up all the services by leveraging the main launch file (all.launch.py). A segment of the robot launch file is shown below:

```
def generate_launch_description():
    pkg_hermes_robot_description = get_package_share_directory('hermes_robot_desc
ription')

    # Convert hte corrected.urdf.xacro file to a urdf file
    xacro_path = os.path.join(pkg_hermes_robot_description, 'urdf', 'corrected2.u
rdf.xacro')

    # Convert the xacro file to a urdf file
```

```
    result = subprocess.run(['xacro', xacro_path], capture_output=True, text=True
)
    urdf_xml = result.stdout

    # Save the urdf to a file to look at later
    urdf_path = os.path.join(pkg_hermes_robot_description, 'urdf', 'corrected.urd
f')
    with open(urdf_path, 'w') as outfp:
        outfp.write(urdf_xml)
```

WRITING A WORLD FILE

The world file is a critical component to validating the sensors on a robot. It is the first baseline you will use to determine if your algorithms are accurate. If the world does not reflect your actual operating conditions, you will experience the same "garbage in garbage out" data processing issue the AI community and many others are struggling with. This mistake occurred when we began with a simple, unrealistic world file. It consisted of a room with two obstacles, with an additional small room that was meant to act as a hallway. This worked great for initial testing. However, as the project progressed, it was noted that points within the laser scan were drifting. Initially, this was assumed to be related to the SLAM, as there were very few features within the simple room to latch onto. Working on this assumption, the world file was updated with a example from the community: a factory floor, with obstacles to avoid, to better match the use case and environment for the robot. You see, since swerve drive robots can move laterally, moving heavy objects without rotating them on a factory floor would be ideal for this robotic system, especially if used in swarm for larger applications. However, even upon changing the world, the drift of the points and subsequent high error on the maps led to the discovery that the odom calculation was incorrect, which resulted in an incorrect TF transform from the base_link to the odom frame. This was corrected, but the point drift is important to note because of this miscalculation. An example of the point drift that occurs when the odometry or any other parent TF is shown below. Note the improved alignment of the point cloud:

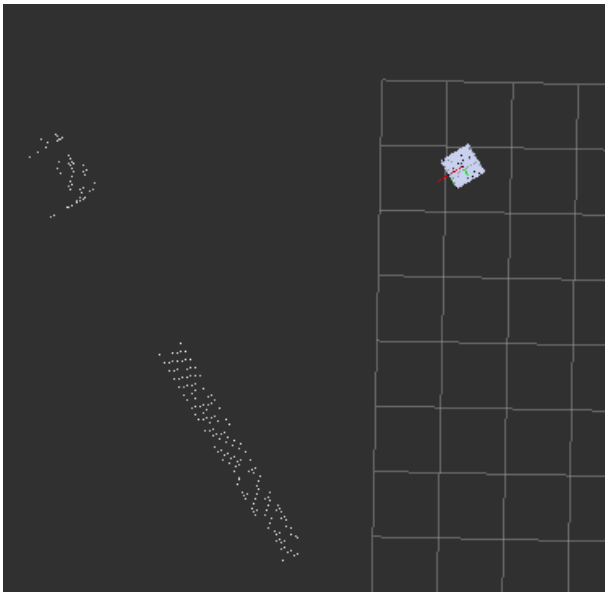


Figure VIII: Misaligned points due to incorrect odometry

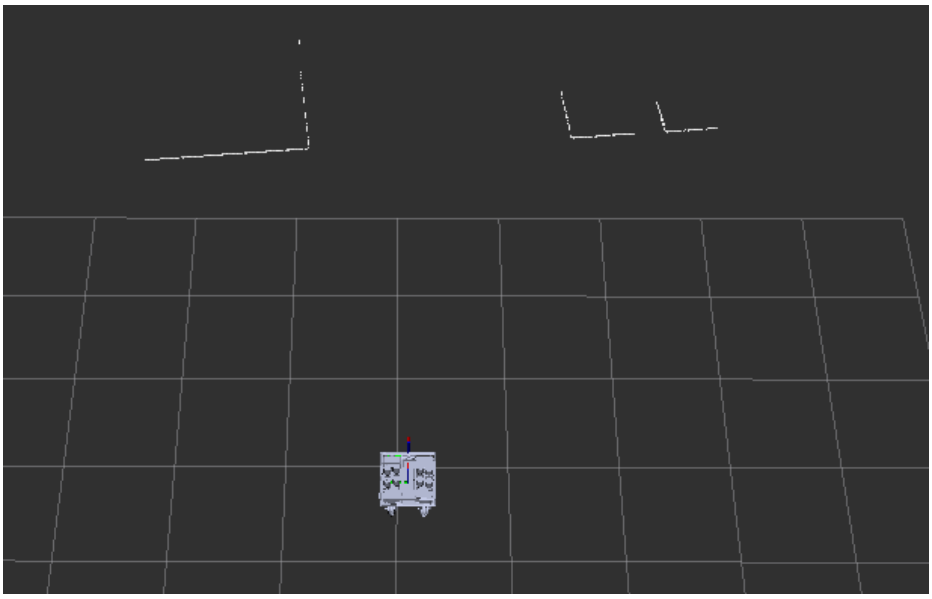


Figure IX: Corrected Odometry leading to Proper Laser Scan

ARCHITECTURE OF THE ROS NODES

Most standard ROS controller nodes for robots reflect the work accomplished for the standard "differential drive" robot controller, simulating a swerve drive robot. This work is contained in [20] and is continually evolving. Normally, these were written in C to provide runtime speed [20]. I believe this is the correct course of action, as it maintains a focus on runtime speed, something critical in robotics. For both readability and for this paper to function as a first draft for a full swerve drive controller, the researcher has opted to use both Python and C++. The former will be great for understanding the work conceptually, while the latter is built for speed on deployment.

Unlike the traditional differential drive robot, the swerve drive robot has four independently driven, each of which can rotate on a radial axis. This requires a unique controller node to manage the movement of the robot. The controller node is responsible for taking in the velocity commands from the user, and converting them into the appropriate wheel velocities through a kinematic model. This model describes the relationship between the wheel velocities and the robot's linear and angular velocities, deriving most of its corpus from the formula for circular motion to define wheel speeds and angles. Upon receiving a command, in the standard convention of the */cmd_vel* topic, the controller node will publish commands to the swerve drive modules, which are themselves ROS nodes. In an attempt to keep the code base clean, the modules run separate computations for PID controls. This is done to ensure that the controller node is not bogged down with computations, and can focus on the primary task of converting the velocity commands into wheel velocities. Additionally, since they are broken out as separate components, each node has its own error topics, which can be monitored by the controller node. This allows for easier debugging and maintenance of the robot, especially when it is in the deployed for use.

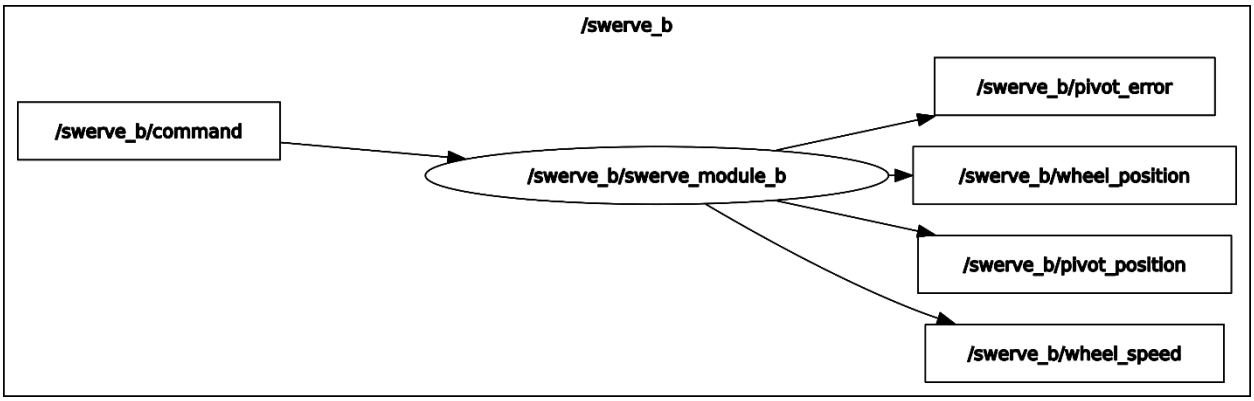


Figure X: A single swerve module node

Additionally, when considering your inputs, the goal of this research was to integrate the NAV2 software stack. A critical observation to be made about the topic publication is that NAV2 has a default */nav_cmd_vel* topic, which publishes an X heading velocity and an angular twist velocity for the z axis. This can be adapted by changing the holonomic configuration within Nav2. For our purposes, to ensure both crab drive and traditional drive is supported, a conversion node was developed to make the robot move in a traditional style. This is done by leveraging the Pythagorian Therom, converting the single X magnitude into an X and Y component (with the original X acting as the hypothenuse). This is then converted into the appropriate wheel velocities. The angular velocity is perpetuated forward, and does not need a conversion. The diagram below illustrates how ROS manages this conversion amongst the controller nodes, taking a */cmd_vel_nav* and using the */converter_node* to convert it to a */new_cmd_vel_nav* topic which the /twist_mux node can consume.

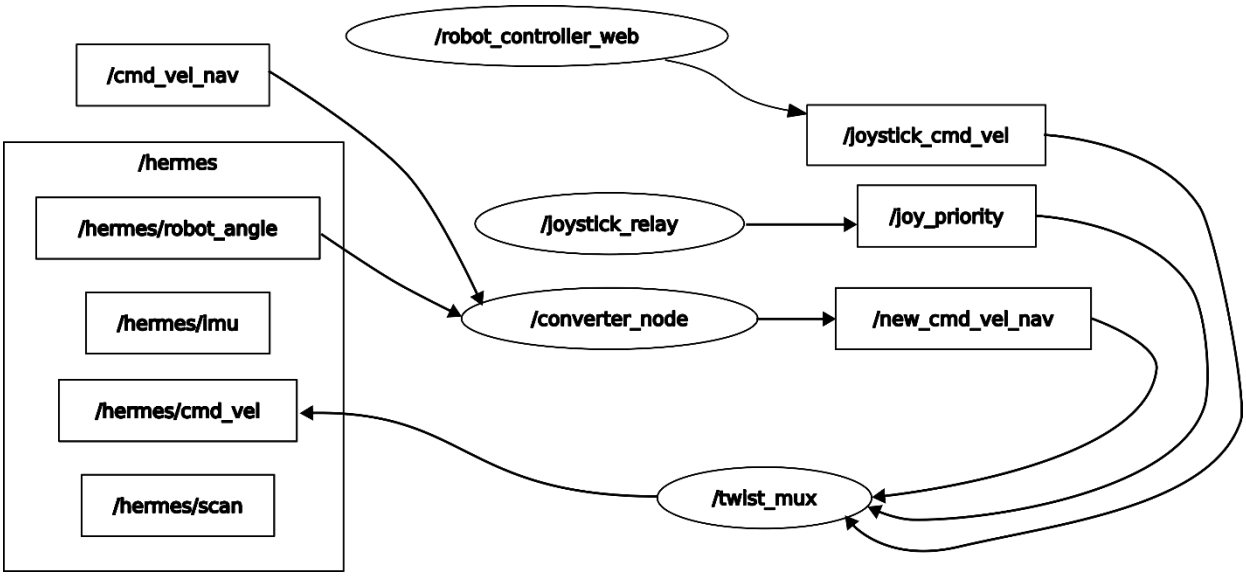


Figure XI: Hermes Control nodes

Furthermore, when considering ease of use and understanding of the robot, a typical control through the keyboard arrow keys or WASD felt insufficient. As such, a webapp was created to

help visualize the commanded wheel angles and velocities through a visual representation of the robot. It also includes a slider element to command linear and angular velocities. This is a critical step, as it allows the user to understand the robot's movement in a more intuitive way and allows for easier debugging of the robot's movement.

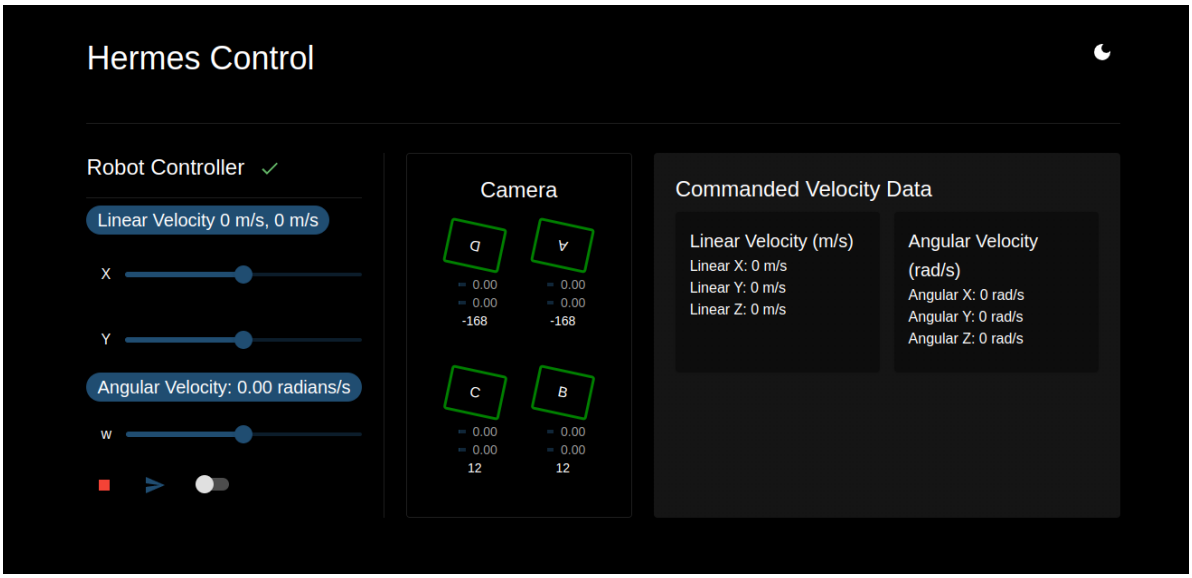


Figure XII: An example of the Swerve Drive UI

This controller, however, presented its own set of challenges. Accepting both user input and nav messages meant a topic that was being published to by two separate nodes. To solve this, the twist_mux node, a standard node within the ROS ecosystem, was deployed and configured to manage the topics by prioritizing and consolidating them to a common topic that the robot can ingest. The twist mux can be seen in Figure XI.

COMPUTING ODOMETRY FOR SWERVE DRIVE

The odometry is based on a combination of data pulled from the kinematics of the model as well as the sensor data pulled from the onboard IMU. Odometry is used to compute a position for the robot relative to its starting position. It is then the responsibility of the mapping algorithm to compute the error, and supply a transform that better stations the robot in the physical world. This is useful in the event of wheel slip, IMU drift, or calibration error. Odometry is typically solved through a combination of GPS, IMU, Optical Flow, or other sensors. On the physical robot proposed by this paper, there would be two such IMU: one independent IMU, as well as the additional IMU data pulled from the camera module. In this way, there is redundancy of the sensor suit as is convention within modern robotics. However, for the simulation, we only consider the single IMU positioned at the center of the robot. This is briefly shown in the previous discussion of TF frames. Its data is leveraged to determine the angular position of the robot, which is then used to compute wheel angles and velocities. The positional change of the wheels based on this is used to determine the linear position of the robot within the world.

The following computation is used to determine the wheel angles and velocities given a pair of linear velocities (X and Y) as well as an angular velocity about the Z axis (W).

$$V_{\text{ith caster}} = V_{\text{robot}} + \omega_{\text{robot}} \cdot O$$

From here on, "robot" will be anything with subscript r, "caster linear velocity" will be subscript c, and "offsets" will be O. They will follow such that "ao" would represent the wheel offset of module A, "at" would represent the tire velocity of wheel A, "r" would refer to the robot in its entirety, and r as a constant represents the radius of the wheel. Note that, in this convention, "caster" is being used instead of "wheel" to avoid any confusion between angular velocity ω and the single letter w. The word "tire" was also considered, but that presented a conflict with t, or time

$$\begin{bmatrix} V_{xc} \\ V_{yc} \\ 0 \end{bmatrix} = \begin{bmatrix} V_{xr} \\ V_{xr} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ w_r \end{bmatrix} \begin{bmatrix} O_{xc} \\ O_{yc} \\ w_r \end{bmatrix}$$

Which implies that:

$$V_{xc} = V_{xr} + \omega_r \cdot O_{xc}$$
$$V_{yc} = V_{yr} + \omega_r \cdot O_{yc}$$

We can define the magnitude of the wheel velocity as:

$$V_t = \sqrt{V_{xc}^2 + V_{yc}^2}$$

Which, expanding the above equations, gives us:

$$V_t = \sqrt{(V_{xr} + \omega_r \cdot O_{xc})^2 + \left(V_{yr} + \omega_r \cdot O_{yc}\right)^2}$$

Finally, we can derive the formula for the wheel angle as:

$$\theta_t = \arctan\left(\frac{V_{xr} + \omega_r \cdot O_{xc}}{V_{yr} + \omega_r \cdot O_{yc}}\right)$$

Which means, to keep track of our position over time given these formulas, we can use the following equations:

$$x_r = x_r + V_{xr} \cdot t + \frac{1}{2} \cdot \omega_r \cdot O_{xc} \cdot t^2$$

From equation **#1** and **#2**, we can derive the following equations for the wheel angles and velocities based on commanded robot linear and angular velocities:

$$V_c = \sqrt{(V_{xr} + \omega_r \cdot O_{xc})^2 + (V_{yr} + \omega_r \cdot O_{yc})^2}$$

$$\theta_c = \arctan\left(\frac{V_{xr} + \omega_r \cdot O_{xc}}{V_{yr} + \omega_r \cdot O_{yc}}\right)$$

With the addition of this formula, **#3**, we can then calculate the angular velocity of the caster (wheel) as it relates to the linear velocity of the wheel as well as the radius of the wheel:

$$\omega_c = \frac{V_c}{r}$$

Overlaid on the robot seen from the bottom, the caster wheel angles and velocities are shown below:

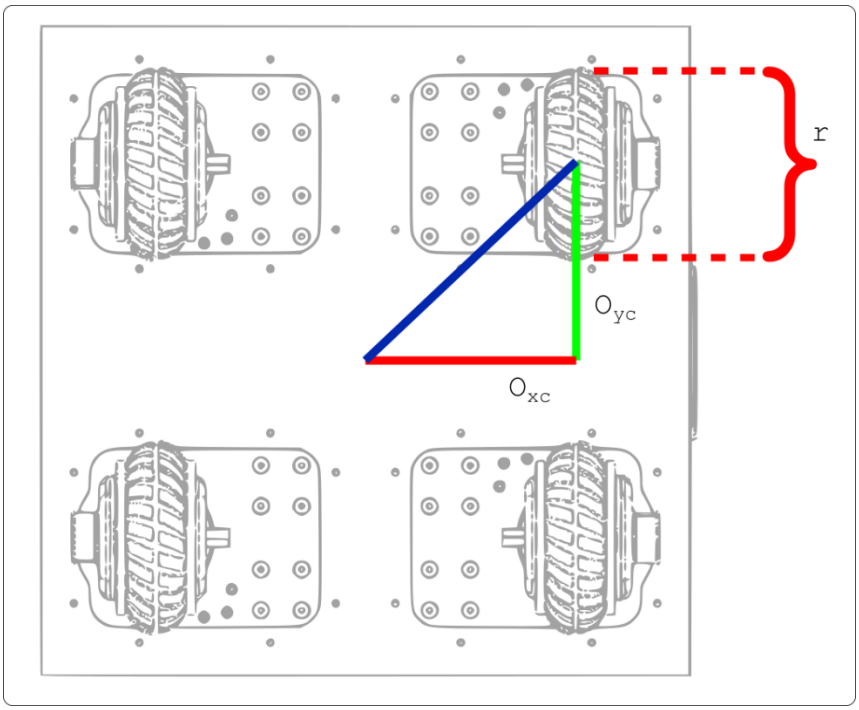


Figure XIII: Diagram of the Swerve Drive Robot with Physical Constants

For a more concrete example, the research team has created an online simulation to illustrate the movement of the robot. This simulation is available through the Forsyth Creations webpage, or citation 27.

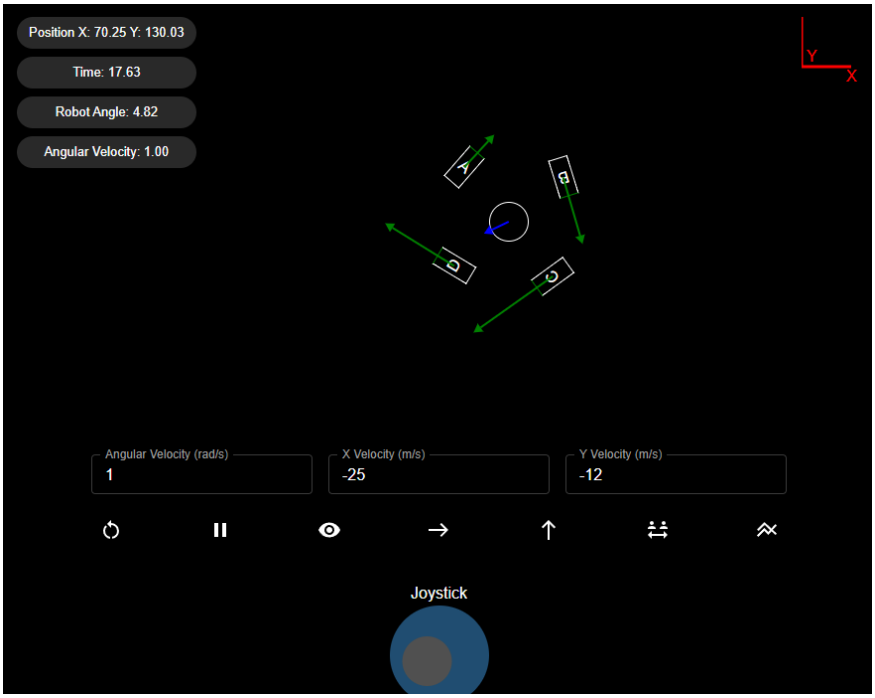


Figure XIV: Online Simulation 1

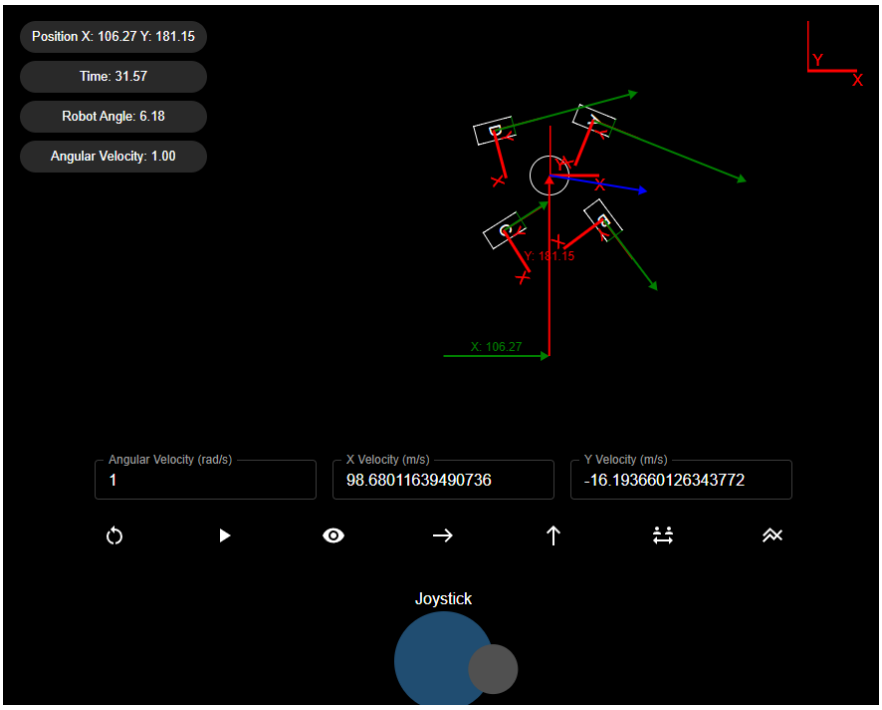


Figure XV: Online Simulation 2 with Coordinate Frames

Within this simulation, you can view a practical example of wheel angles, headings, basic TF transforms, as well as historic data of the wheel positons as a sanity check to ensure the robot is moving as expected.

WRITING A PID CONTROLLER

As stated previously, the control system architecture attempts to keep smart software solutions in mind. One major consideration made within this design was how to better accomidate a flexible software set that is light weight enough to be deployed on a Raspberry Pi or other such device. This consideration is critical for the PID tuning, which has been added within each of the swerve_module nodes. Since the Raspberry Pi 5 (the hardware candidate for this software) lacks a GPU, training a neural network is not a viable option. As such, the researcher has opted to use a PID controller to control the speed of the wheels. This is a common approach in robotics, and is well suited for this application. However, to ensure that the PID controller is tuned correctly, the researcher has implemented a series of tuning methods to optimize the performance of the PID controller. This section will discuss the tuning methods used, and the results of each method.

PID is an acronym for Proportional, Integral, and Derivative. It is a control loop feedback mechanism that is used to control the output of a system. The PID controller calculates an error value as the difference between a desired setpoint and a measured process variable. The controller attempts to minimize the error by adjusting the process control inputs. The PID controller algorithm involves three separate constant parameters: the proportional, the integral, and the derivative values. The proportional value determines the reaction to the current error, the integral value determines the reaction based on the sum of recent errors, and the derivative value determines the reaction based on the rate at which the error has been changing. The weighted sum of these three actions is used to adjust the process via a control element such as the position of a control valve or the power supply of a heating element.

For our purposes, each wheel is tuned based on the torque specification of the motor, and the moment of intertia of the wheel. Three tuning methods were attempted for this robot system: a standard Zieger-Nichols tuning method, a manual tuning method that attempts all possible values within a range, and a gradient decent method used to optimize not only the parameters themselves, but make the calcualtion faster. The outputs of the PID controller tuning are shown below. It should be noted that, unless otherwise stated, the goal of each of these tuning simulations is to take an object from 0 m/s to 5 m/s as fast as possible, minimizing overshoot, minimizing oscillation, and minimizing the time to convergence.

To accomplish this tuning, the researcher used a simulation of the swerve drive robot. This simulation was run in Python, accepting a max_torque, moment of inertia, a desired speed, an acceleration dampener (since we assume the robot is not a perfect system), and an ideal convergence time. This convergence time will be critical when we discuss the gradient decent PID tuner, later in this paper. The simulation runs a simple Euler integration loop, where the torque is applied to the wheel, and the wheel accelerates based on the torque and moment of inertia. The simulation runs until the wheel reaches the desired speed, or until the maximum time is reached. The simulation also records the time it takes to reach the desired speed, as well as the overshoot and oscillation of the wheel speed. This data is then used to tune the PID controller.

As such, the following constants were applied in simulation. These values were selected based on the specifications of the motors, as well as the moment of inertia of the wheels:

```
accel_damping: 0.5
moment_of_inertia: 0.4
max_torque: 3.75
time_frame: 10
dt: 0.1
```

Zieger-Nichols Tuning

Using the Zieger-Nichols tuning method [21] was applied to the swerve drive modules. The results of this tuning are shown below:

Controller	P	I	D
Zieger PID	0.6 * Ku	2 / Tu	Tu / 8

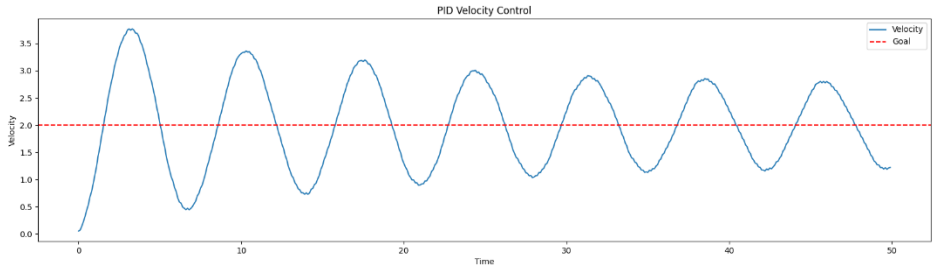


Figure XVI: Zieger Graph

Notice how the Zieger model does not converge within this timeframe. Instead it appears to oscilate, making it an unideal controller for this scenario. This is a common problem with the Zieger-Nichols tuning method, as it is not designed for systems with high inertia or systems that require fast response times. As such, this method was not used for the final tuning of the PID controller. This paper, and the tuning methods therein, is seeking a reliable, fast, and accurate tuning method that can be used in a variety of situations.

MANUAL TUNING GROUND TRUTH

So instead, the researcher attempted to manually tune the PID controller. This was done by iterating through a range of values for each of the PID parameters and selecting the combination that provided the best performance. The results of this tuning are shown below. The following are the ranges that were used for the tuning:

```
kp_range = np.linspace(0.01, 200, 30)
ki_range = np.linspace(0.01, 200, 30)
kd_range = np.linspace(0.01, 200, 30)
```

This yielded what is known as a "brute force" tuning method. The results of this tuning are shown below, and is used as a ground source of truth as we compare it to other methods:

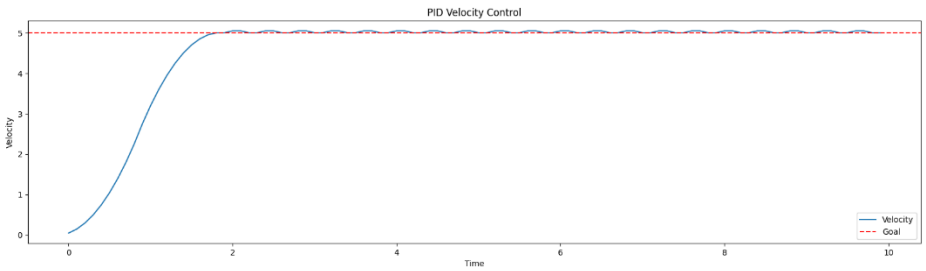


Figure XVII: Manual tuning within select range of values

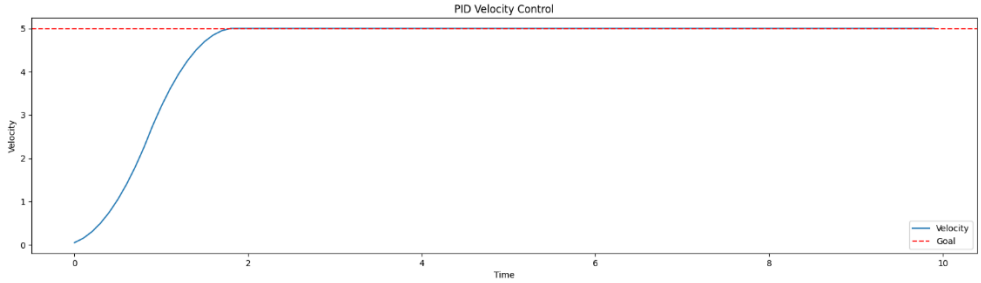
Notice, however, that there are a few issues with this method. First, the computation alone is very slow, as it iterates through a large number of combinations. For this range of values, the program took one minute to execute. This is not suitable for our tuning needs, as it is not fast enough to be used in a real-time system. This method, running on discrete changes to the P, I, and D values has another problem: inability to effectively process out the remaining oscillations. As you can see, the oscillations persist as the graph proceeds to the fixed point. As such, this method was not used for the final tuning of the PID controller. Instead, the researcher attempted to use a gradient decent method to tune the PID controller. Note that, for this method, the convergence time (calculated by taking an average of five executions and determining if it was settling out to a value) was two seconds. The importance of this will be discussed within the gradient decent section. Also, for the sake of completeness, the following are the values that were used for the tuning, as well as the overshoot and oscillations that were observed:

```
kp = 27.59
ki = 0.01
kd = 13.802
overshoot = 0.02
oscilations_within_timeframe = 20
```

Note that we calculate oscillations by determining any peaks or valleys that deviate beyond a threshold of 0.02 from the goal.

GRADIENT DECENT TUNING

The gradient descent method is an advanced optimization technique used to fine-tune PID parameters by minimizing the error between the desired and actual system outputs. It iteratively adjusts parameters in the direction of the steepest descent of the error function, calculated as the gradient with respect to each parameter. By selectively defining the cost function, this method allows prioritization of specific performance metrics, such as reducing response time, oscillations, or settling time. The time of calculation was also significantly reduced, taking five seconds to process proper tuning values. The results of this tuning are shown below:



This method converged within two seconds, consistent with the brute force method above. Additionally, the oscillations are reduced to zero. To further validate this, researchers used the same PID parameters to try and govern the motion across multiple goal states. The following are the results:

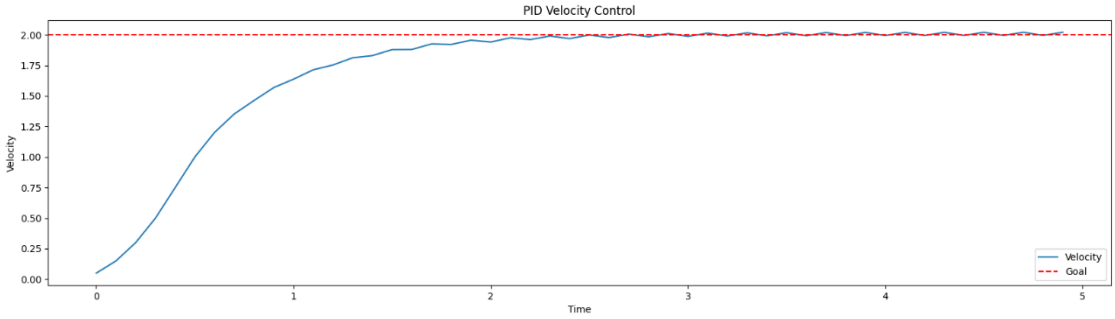


Figure XVIII: Standard PID from 0 to 2 m/s

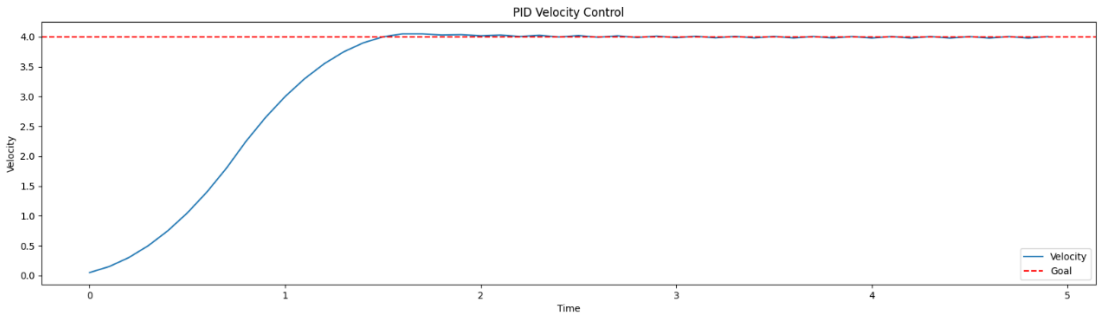


Figure XIX: Standard PID from 0 to 4 m/s

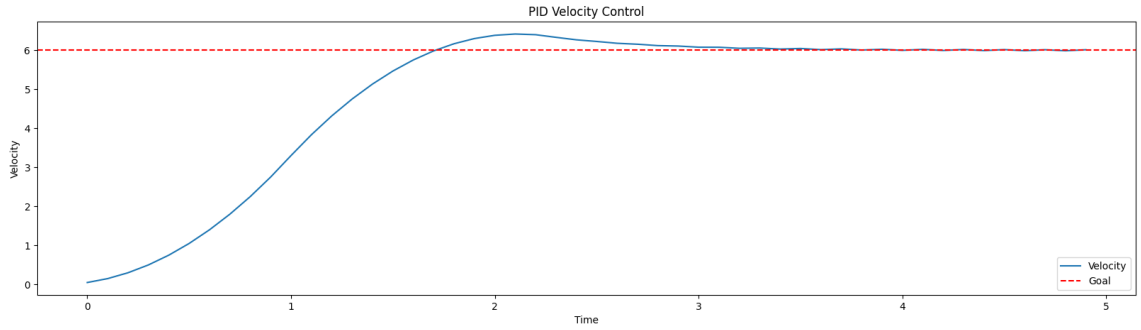


Figure XX: Standard PID from 0 to 6 m/s

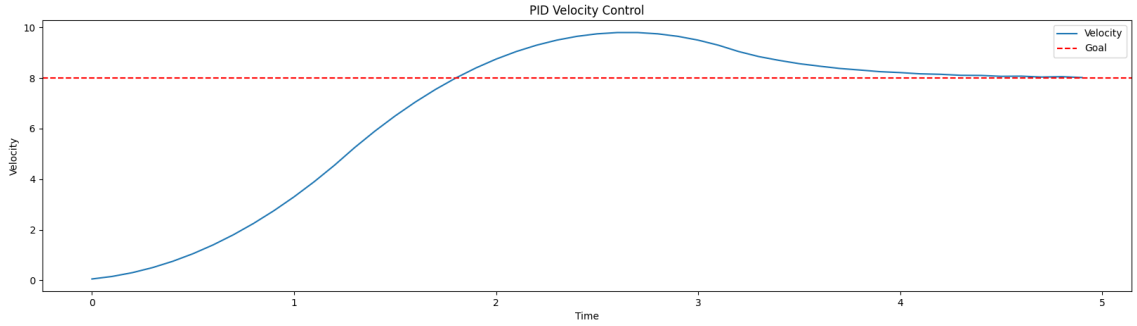


Figure XXI: Standard PID from 0 to 8 m/s

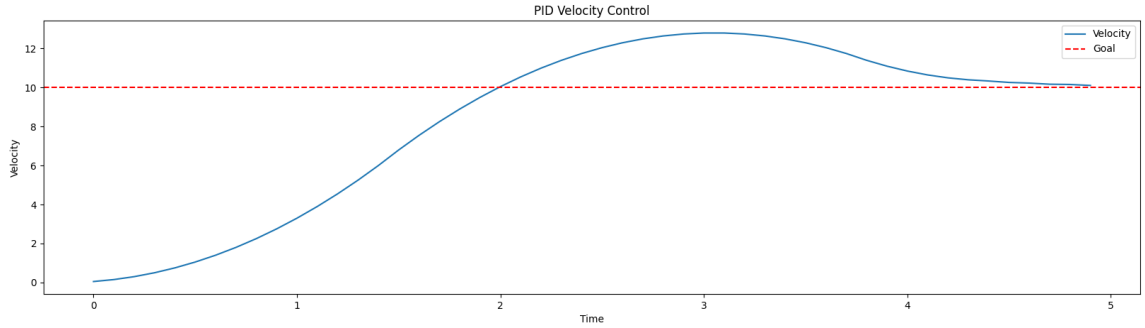


Figure XXII: Standard PID from 0 to 10 m/s

Note the following results for these tests:

Goal (m/s)	Time to Convergence (s)	Overshoot	Oscillations	Error	PID Values
2	1.9	0.02	10	13.83	(1.54, 0, 0.037)
4	1.8	0.05	8	32.16	(1.54, 0, 0.037)
6	3.2	0.41	18	61.97	(1.54, 0, 0.037)
8	4.7	1.8	20	112.79	(1.54, 0, 0.037)
10	5.1	2.8	18	167.78	(1.54, 0, 0.037)

With this data, we notice the trend of increasing overshoot and oscillations as the goal increases. As the error is, in part, a side effect of these metrics and not normalized based on the increased goal difference, we include it in the table for completeness but will not consider it for this discussion. Changing parameters in this manor is a common problem with PID controllers, as

they are not designed to handle a large range of change without considerations being made for the gain. The datapoint for the 10 m/s is especially troubling, as this overshoot is far beyond standard error. To reduce error, the researcher thought to use reinforcement learning. However, since this work is bounded to low-level hardware, and most reinforcement learning algorithms are designed to run with CUDA or other such technologies, the decision was made to discretize and multi-train the PID controller. This was accomplished by designing a class that takes a parameter for the range you expect the robot to operate in. It will then train subtle variations of the PID controller, gaining the PID values for each variation. These are stored in a dictionary that can be accessed in O(1) time or, in the worst case, a 0(N) time complexity, making it ideal for real-time applications. This class is as follows:

```
class MultiPID:
    def __init__(self, difference, step = 1):
        self.difference = difference
        self.step = step
        self.tunings = {}
    def train(self):
        for i in range(1, self.difference, self.step):
            # Train a model based on the spec, using i as the goal
            print(f"Training for goal {i}")
            pid = NewPID(intertia=moment_of_inertia, max_torque=max_torque, loggi
ng=True, accel_damping=accel_damping, ideal_convergence_time = 1)
            pid.reset()
            params = pid.auto_tune_gradient_descent(i, max_iter=200, max_time = t
ime_frame, learning_rate=200, auto_accept=True, tolerance = 0.001)
            self.tunings[i] = params
    def get(self, current, goal):
        # Get the PID values for the goal
        int_difference = int(abs(current - goal))
        tuning = self.tunings.get(int_difference, None)
        if tuning is None:
            # Find the closest tuning
            closest = min(self.tunings.keys(), key=lambda x:abs(x-
int_difference))
            print(f"Using closest tuning for goal {goal}: {closest}")
            tuning = self.tunings.get(closest)
        return tuning
multi = MultiPID(15)
multi.train()
```

Note that, in cases where the exact discrete case isn't available, the closest tuning is used. Also understand that, to reduce complexity at run time, it is assumed that a movement from 4.5 to 4.8 m/s, or from 3.3 to 3.5 m/s are identical based on the difference being on average 0.25. The algorithm will simply return the lowest PID values from the dictionary. The results of this multi-PID approach are shown below:

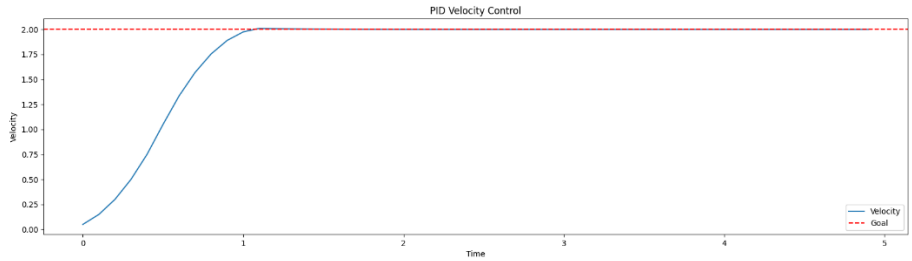


Figure XXIII: Discrete PID Control: 0 m/s to 2 m/s

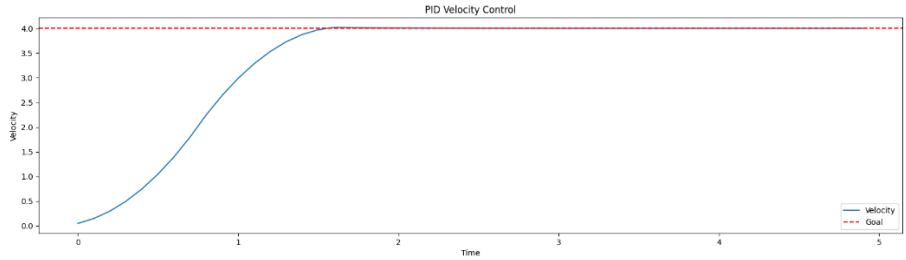


Figure XXIV: Discrete PID Control: 0 m/s to 4 m/s

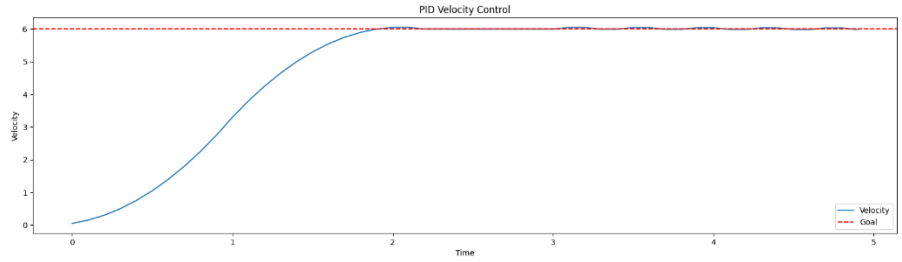


Figure XXV: Discrete PID Control: 0 m/s to 6 m/s

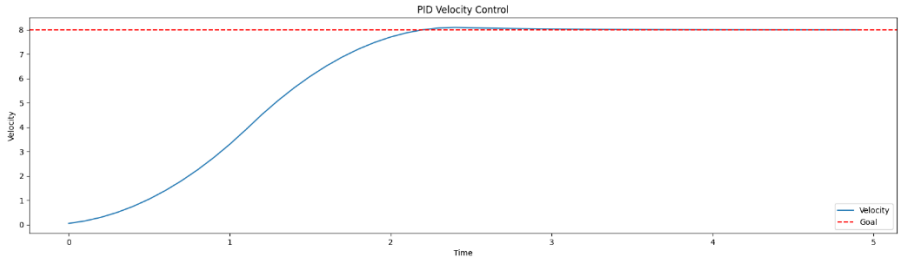


Figure XXVI: Discrete PID Control: 0 m/s to 8 m/s

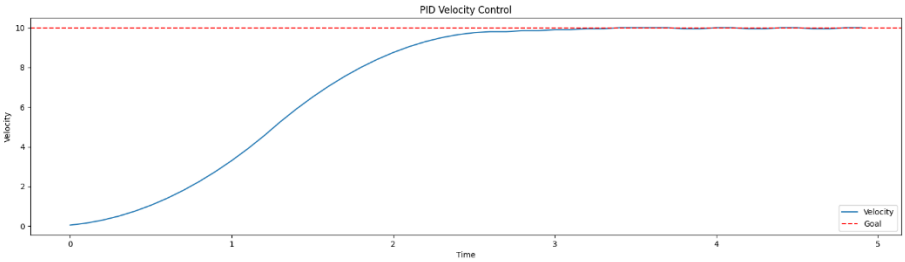


Figure XXVII: Discrete PID Control: 0 m/s to 10 m/s

Note the following results for these tests:

Goal (m/s)	Time to Convergence (s)	Overshoot	Oscillations	Error	PID Values
2	1.2	0.01	1	10.71	(1.2, 0, 0.0)
4	1.8	0.02	4	31.82	(0.812, 0, 0.0)
6	2.2	0.05	11	60.10	(41.72, 0, 23.01)
8	2.5	0.10	15	93.52	(0.608, 0, 0.0)
10	3.3	0.00	0	133.9	(27899.53, 0 , 19422.90)

Notice that, in every case comparing the single PID tune to the multi-tune, the overshoot and oscillations improved. This is consistent with typical control systems. Typically, convergence is a function of an aggressive P term. However, aggression must increase with variance in difference from **current** to **goal**. A single tune will not be able to accommodate such variance. However, this discrete multi-tune method has proven highly effective for this task. It resembles the harmonic tuning commonly used in robotics, especially on Bambu Labs 3D printers.

When put into practice within simulation, the wheel speed and pivot angle performed within similar ranges. The following graphs illustrate this parity. You will find that the graphs illustrate a publisher/subscriber model, where the main controller node requests a position (ie */swerve_d/rqst_wheel_speed*) and the modules processes the request through the PID tuning controller and publishes the result to the */swerve_d/wheel_speed*. When a HAL (Hardware Abstraction Layer) or some such variant could then be written to convert these commands into physical signals for the motors of a physical robot.

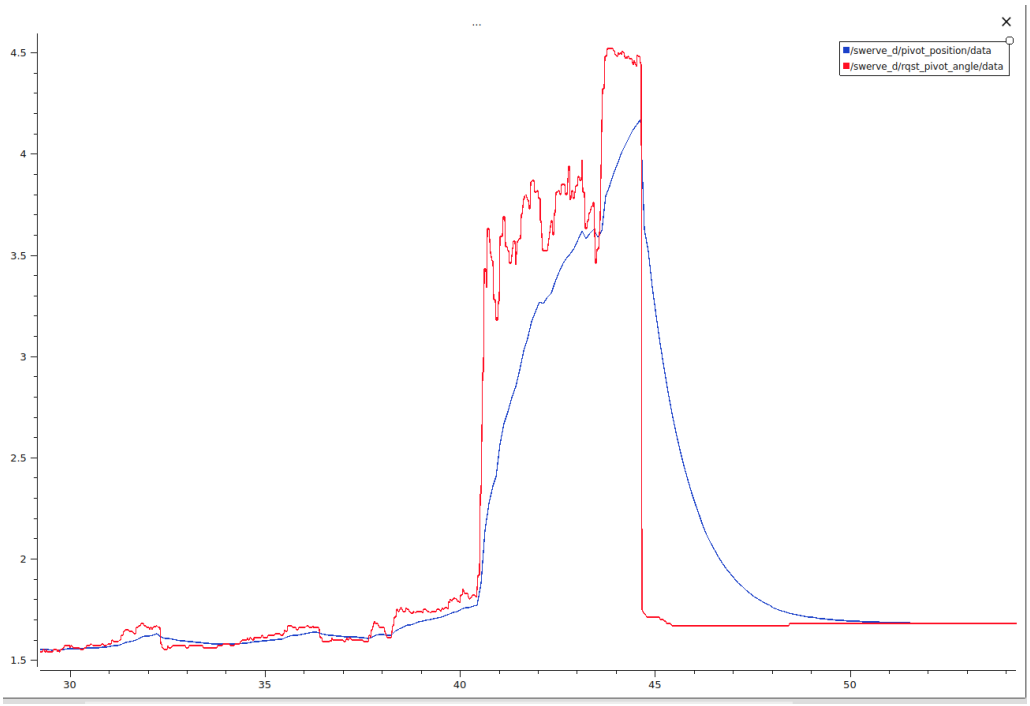


Figure XXVIII: PID Controller of Pivot Angle

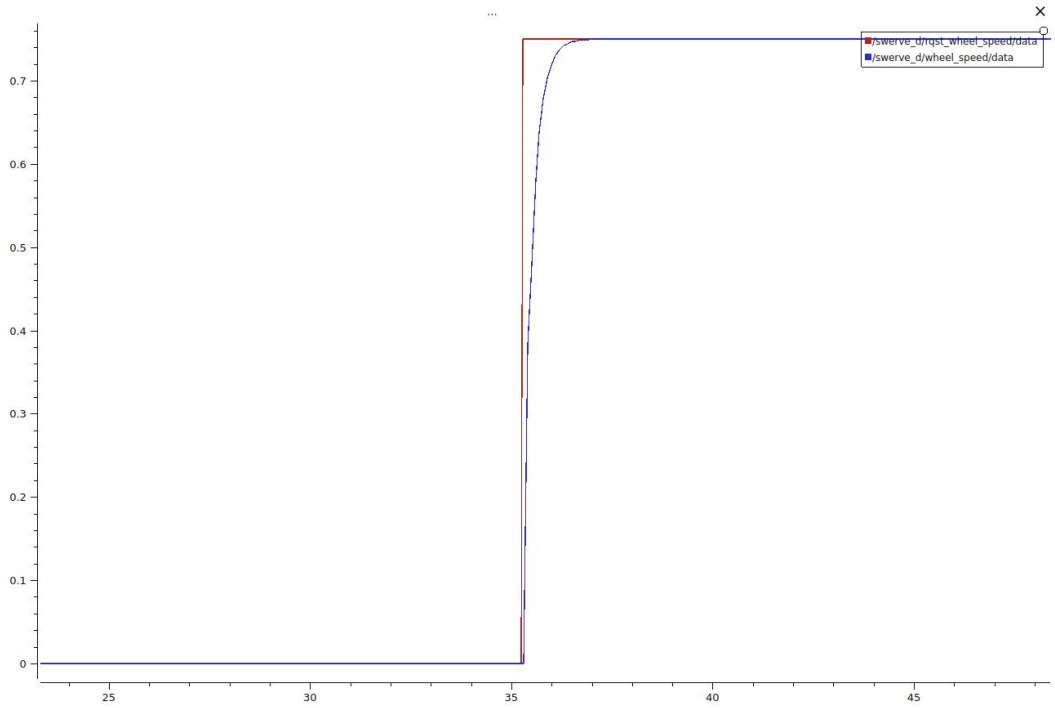


Figure XXIX: Wheel Speed PID Example

When it comes to open-source perception and navigation systems, the researcher found Nav2 was the best candidate to integrate. While it has capabilities to use voxel grids or 3D SLAM algorithms for navigation, the researcher favored a simple 2D spoofed laser scan approach. This keeps the perception algorithm simple and favors lightweight hardware. This decision is not without its drawbacks, namely the fact that a lower-fidelity perception system will result in lower-fidelity outcomes for interfacing with the environment. However, since this paper focuses on an overall schema for ROS2 development within Solidworks, Gazebo, and then eventual deployment to a Raspberry Pi, this limitation is accepted. Deployment on better hardware, such as a Jetson, would come with less restrictions and recommendations by this research to leverage more sensors/better perception algorithms to increase robotic situational awareness.

That being said, what we have elected to implement is a spoofed laser scan. This amounts to taking a single line of pixels from a depth scan image (provided by an Intel Realsense) and generating a laser scan topic based on the perceived depth map. This allows seamless integration with the existing slam_toolbox software suite, which is designed to generate a map out the box by ingesting a laser scan topic. Instead of designing a custom node, the depthimage_to_laserscan package was leveraged to create the laser scan data needed for slam_toolbox.

Following this, Nav2 is directed to subscribe to the appropriate laser scan and slam_toolbox topics, and will generate a cost map based on what it perceives are the obstacles. An example of this cost map is shown below, overlaid on the existing floor of the room to demonstrate accuracy in the perception.

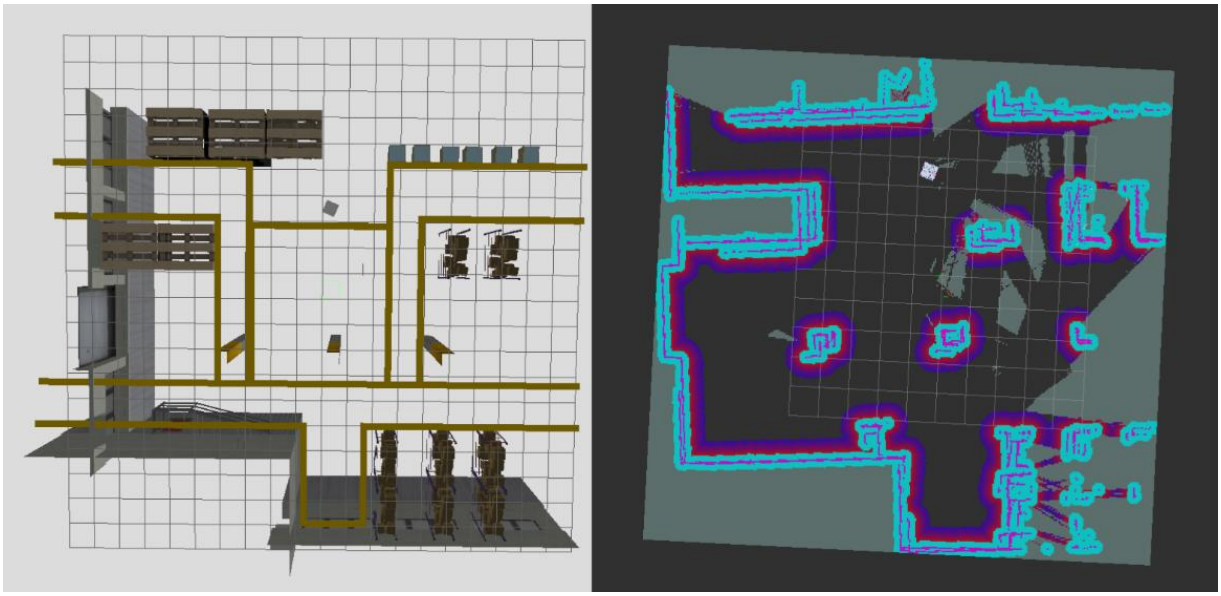


Figure XXX: Ground Truth and Cost Map

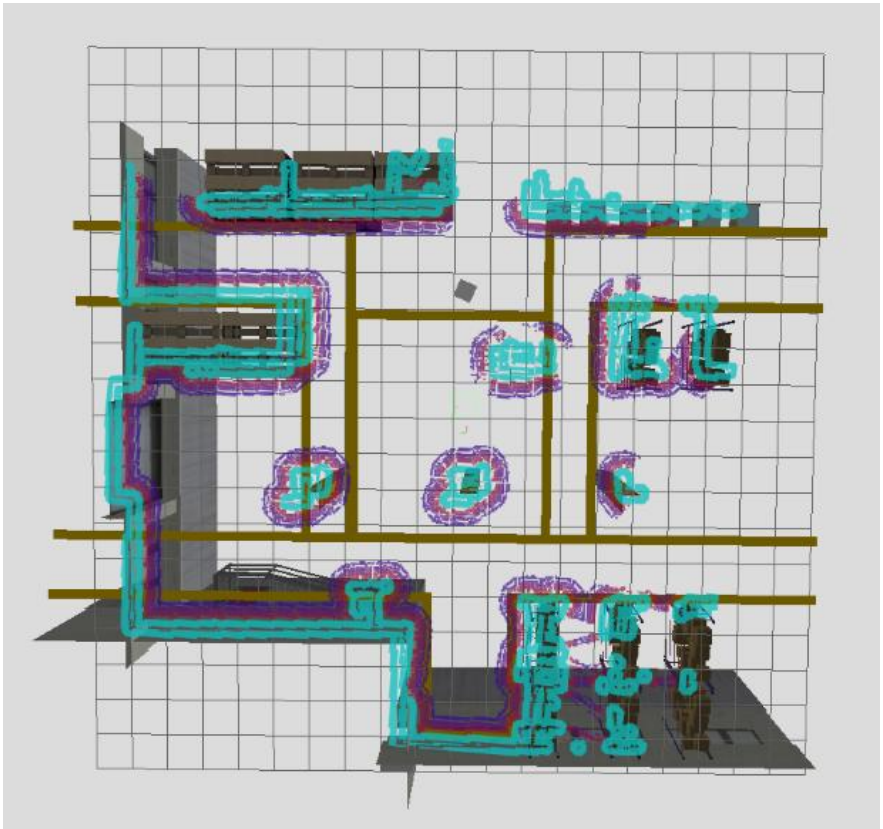


Figure XXXI: Costmap overlaid on ground truth

RESULTS

POSITION VALIDATION

When validating a localization system, industry convention is to use a secondary system to validate the first. This parallels the concept of redundancy within a robotic system, where one hardware system can act as an enforcement or check of the other system. In simulation, specifically for this problem, the ideal way to validate that your robot's absolute position and perceived position are identical is by cross referencing the Gazebo model's position with the pose of the robot.

In the Gazebo environment, this data is made readily available in the component tree. It was the assumption of the researcher that this data would also be available to the ROS environment when the joint transforms were mapped from Gazebo to ROS2. This was unfortunately not the case, as the mapping function (conversion from Gazebo to ROS2) only persisted data from the children of the root node (in this case, the hermes_robot_description). However, to validate the system, the absolute pose data of the root note was needed. Toa achieve this, the following topic was mapped from Gazebo to ROS2:

```
- ros_topic_name: "/hermes/gazebo/pose"
  gz_topic_name: "/world/demo/dynamic_pose/info"
  ros_type_name: "geometry_msgs/msg/PoseArray"
  gz_type_name: "gz.msgs.Pose_V"
  direction: GZ_TO_ROS
```

Then, analyzing the subsequent position array made available to ROS, you can identify the index of the pose that represents the absolute position of the *hermes_robot_description* (which, in our case, is simply the robot) within the world. You can then persist this data to a single PointStamped topic, where it can be used to graphically validate the position and orientation of the robot compared to the goal pose.

A segment of the code to run the conversion from the node within this body of research is presented below. It should be noted that, if the world file changes, that the same index to gain this absolute value will not be guaranteed.

```
def callback_make_point(self, msg):
    # Convert a pose array to a single x and y point
    if not msg.poses or len(msg.poses) <= self.position_index:
        self.get_logger().warn("Received empty PoseArray.")
        return

    # Extract the first pose from the array
    pose = msg.poses[self.position_index]

    # Create a Point message
    point_msg = PointStamped()
    point_msg.point.x = pose.position.y
    point_msg.point.y = -pose.position.x
```

```
    # Add in the timestamp
    point_msg.header.stamp = self.get_clock().now().to_msg()

    # Publish the Point message
    self.publisher.publish(point_msg)
```

Inspiration for this method was derived from the line-error method mentioned in citation #5. Following the conversion, the absolute position will be available on the configured topic. Since we are using PlotJuggler for timeseries analysis, the only other pitfall was ensuring that the */goal_pose* (a topic produced by Nav2) persisted in its data instead of making a single publication at the time of request. Not knowing the functionalities of the navigation server, it was ill advised to change this topic directly. Instead, a "clamping" or "latching" node was written to perpetuate the topic data for better time series representation. A segment of this code is shown below:

```
def listener_callback(self, msg):
    self.new_msg = msg
    self.get_logger().info(f'Received new value: {msg}')
def publish_latest_value(self):
    if self.new_msg is not None:
        if hasattr(self.latest_value, 'header') and hasattr(self.new_msg.header,
'stamp'):
            self.new_msg.header.stamp = self.get_clock().now().to_msg()

        self.publisher.publish(self.new_msg)
```

Armed with a better goal pose topic, */clamped_goal_pose*, and a proper absolute value position, */robot/position/absolute*, the robot's relative position was assigned in RVIZ, as well as a goal pose. The projected path is shown below:

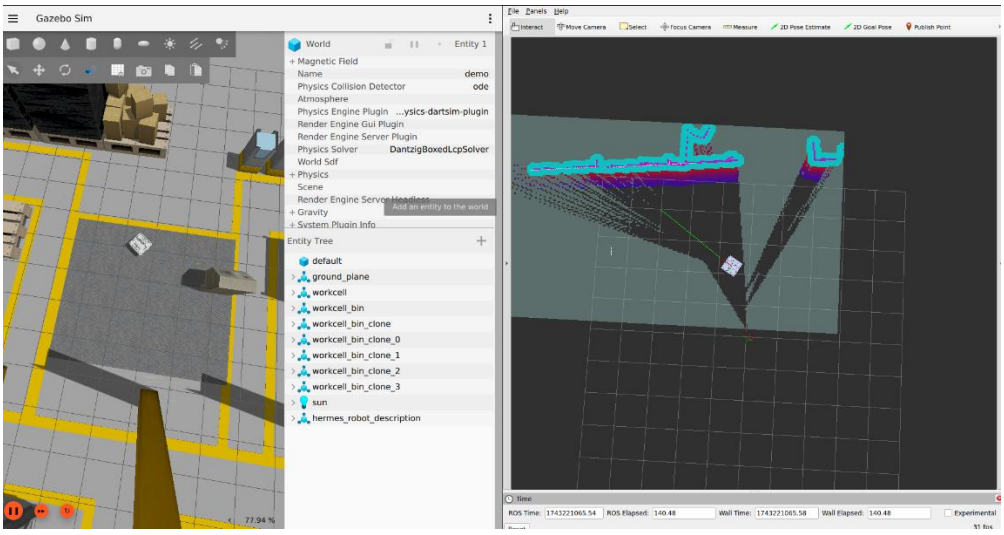


Figure XXXII: Example Path Planning using Nav2

With the subsequent Cartesian change in position shown below:

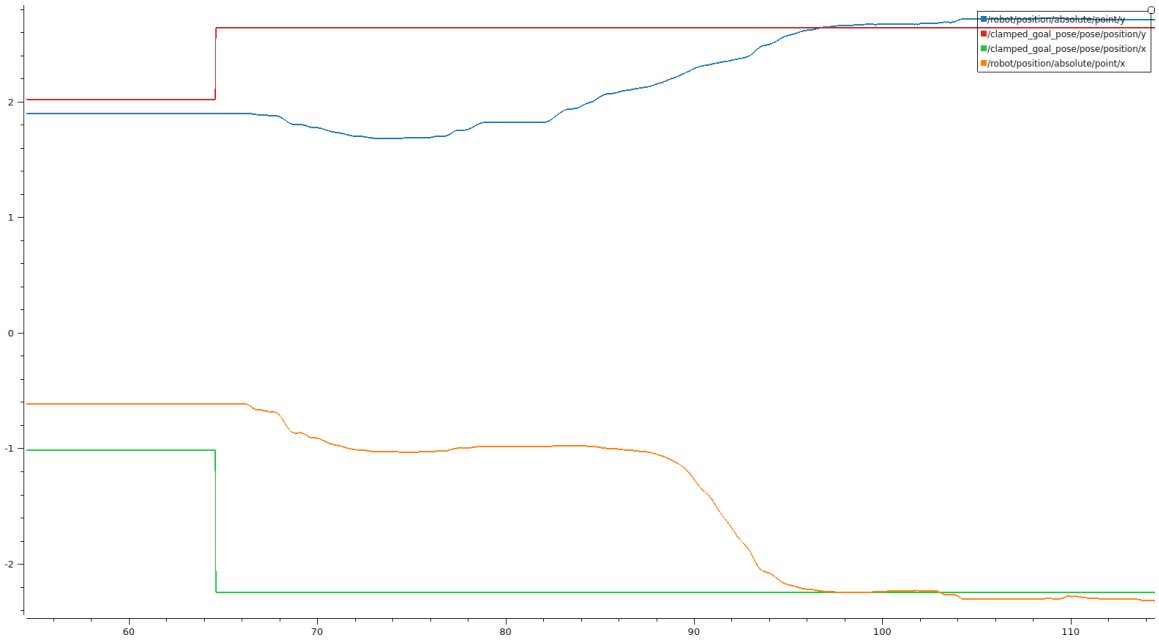
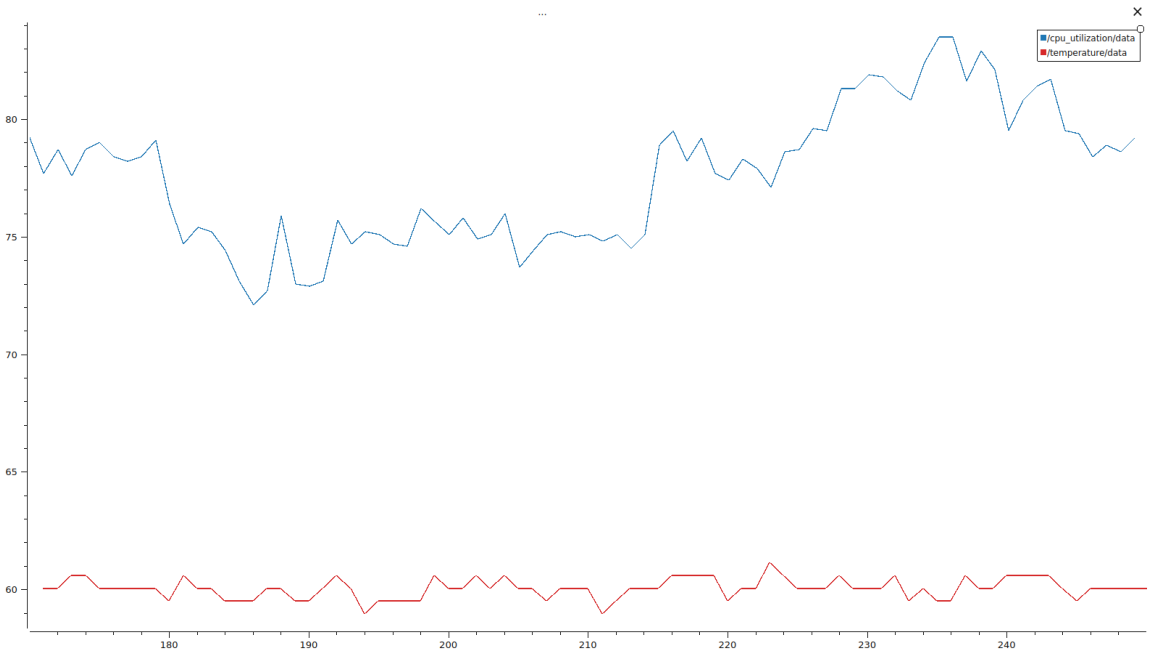


Figure XXXIII: Absolute Position Reached

Note how the PID control system, working together with Nav2, resolves the robot to the indicated position. Any error in position is as a result of odometry to map TF drift, which is common on systems like this. Note that the error on both is below 0.06 of a meter, or about 2 inches. Given the simplified perception algorithms chosen, this further validates the control system.

COMPUTATIONAL HEADROOM: CPU UTILIZATION

To conduct and validate the system performance, it was necessary to ensure that the system was able to run on a Raspberry Pi 5. However, as a body of future work is integrating this simulated controller with a physical robot, there was no existing full test bed. As such, we created a networked simulation of the robot, where the controller node was run on a Raspberry Pi 5, and the rest of the system was run on a standard PC, similar to the setup in [10]. This allowed for a more realistic test of the system, as it would be running in a simulated real-world environment. It also ensures that the simulation is not eating up valuable CPU and RAM resources on the Raspberry Pi 5, skewing the results of the test. As further metrics, the CPU and temperature were exported to ROS topics and then plotted while the robot is run in simulation. With the Pi running Nav2, slam_toolbox, the controller node, and the swerve drive modules, the following results were observed:



Note that the recommended operating temperature of the Raspberry Pi 5 is between -20 C to 85 C [28]. Even under load, this software strategy maintains an average of 61 C, with an average compute load of 80% under full stress.

To further explain this deployment strategy, the following segment of the launch script is provided:

```
    # Add Actions Based on Simulation Mode
    if simulation_mode_value in ['world', 'all']:
        print("Starting sim with world actions")
        ld.add_action(web_controller)
        ld.add_action(robot_description)
        ld.add_action(bridge_launch)
        ld.add_action(world_launch)
        ld.add_action(rviz_launch)

    if simulation_mode_value in ['robot', 'all']:
        print("Starting sim with robot actions")
        ld.add_action(robot_additional_nodes)
        ld.add_action(hermes_controller)
        ld.add_action(nav_updater)
        ld.add_action(temperature_node)
        ld.add_action(cpu_utilization_node)
```

Note that, in the case of the above, the “world” is run on a 5700 Ryzen series CPU, where the “robot” is computation is run directly on the Pi for validation.

DISCUSSION

This paper opened with a slew of questions and interests in developing a swerve drive system that could be deployed on a Raspberry Pi, as well as simulation hardware, with minimal variation between the environments. The intention was to make it easy to deploy to mid-weight hardware, across both ARM and AMD compute architecture, paving the way for further research into ROS2.

Its first focus was to find a way to design and simulate a swerve drive robot in Gazebo, and validate a new ROS controller for the system. This was accomplished by extracting the absolute position of the simulated robot from the Gazebo environment, and comparing it to our perceived map position. This was all made possible by the Gazebo/ROS bridge, which mapped pertinent topics to and from the simulation environment for easy use.

The seconds and third focus of this paper was ensuring sufficient device headroom on a Raspberry Pi, and analyzing what those sensor choices mean for the navigation system. This was accomplished by using C++ over python to result in higher runtime efficiency. This paper also preferred a lighter weight algorithm to keep CPU utilization within bounds, leveraging a 2D laserscan pulled from a depthmap image from a simulated Intel Realsense.

This decision led to trade offs. True, the system remained in bounds of the computational requirements. However, a 3D awareness of the space would promote a better SLAM, and as a result less error in resultant mapping. Forgoing a puck lidar also resulted in an inability to leverage a swerve drive system's greatest gift: on demand directional command. This is recommended for future work and study.

The fourth and final question focus we had was determining the best metrics to pull error from. In a perfect world, as systems like this transfer to high-parallelized processors (GPU), you can deploy ML models specifically trained to govern the motion policies of the machine. As such, to support these models, we identified both the pivot and drive speeds as sources of error that could be tracked and used for ML training. These were given their own ROS topics in hopes of leveraging this system within a different simulation environment in the future.

CONCLUSION

At the end of this work, it's clear that swerve drives offer a powerful and flexible motion system—one that's often overlooked due to their complexity, but increasingly viable thanks to tools like ROS2 and accessible hardware like the Raspberry Pi. With a working driver now available and validated both in simulation and on-device, we've created an entry point for others to start building on top of this system. It's no longer a question of *if* swerve drives can be made to work on constrained systems with ROS—it's how we can now use them to solve more interesting problems. This opens the door for researchers to treat the swerve platform as a baseline, not a barrier.

At the same time, we're clearly moving toward a future where much of this decision-making will be offloaded to machine learning models. That's where this project also tried to look ahead—by exposing low-level control data like pivot and drive speeds and their related error, we've made it easier to plug in learning systems without rewriting the core driver. ML will no doubt absorb much of this workflow in the coming years, shaping everything from motion policies to error correction.

But as we lean into that shift, we also need to stay grounded. There's a famous saying: *when all you have is a hammer, everything looks like a nail.* Right now, AI is the hammer—and its children are the nails. But we can't forget that screws still exist. That is to say: just because we *can* throw a neural net at something doesn't mean we *should*—especially on low-power systems where every clock cycle counts. This work tried to honor that balance: enabling smarter systems, without losing sight of the constraints like power consumption for the compute that will surely plague robotics going forward.

In the end, this project wasn't just about building a robot. It was about building a bridge—between simulation and hardware, between traditional control and machine learning, and between what's possible today and what might come next.

SUMMARY

This project set out to create a swerve drive system that could run cleanly across both simulation and real-world environments, with minimal friction between the two. The goal was to make swerve drive more accessible—not just as a concept, but as a deployable system that runs well on limited hardware like the Raspberry Pi. Using Gazebo and ROS2, we built and tested a robot capable of true swerve motion, validating its accuracy through simulated ground-truth data and map comparisons.

We kept things efficient by using C++ and lightweight sensing, trading full 3D awareness for real-time performance. Along the way, we made conscious decisions about what matters most in constrained systems—and left open hooks for smarter systems down the line. In particular, we flagged key motion metrics for use in future ML models, helping bridge the gap between rule-based control and learned behavior.

The work acknowledges where swerve drive shines, where it still struggles, and where it can grow. More importantly, it lays down infrastructure for future research to build on—be it in SLAM, control policy, or low-power autonomy. With a working driver now available, researchers and developers have a platform to experiment, extend, and evolve.

For future work, the recommendation would be to pursue this approach with better hardware. The NVIDIA Jetson comes to mind, simulating the system in Issac Sim or Issac Lab and comparing the performance between the mid-level hardware to the high-end hardware. Advanced robotics seems to favor compromise between the low and high power, so using high end hardware and software solutions to inform the middle end would be instructive for future development.

CITATION

NOTE: THESE WILL BE CONVERTED TO PROPER CITATIONS SOON, THIS IS MAINLY JUST A DRAFT

1. LR1: https://ieeexplore.ieee.org/document/10063871
2. LR2: https://ieeexplore.ieee.org/document/10307118
3. LR3: https://ieeexplore.ieee.org/document/10242502
4. LR4: https://ieeexplore.ieee.org/document/9752654
5. LR5: https://ieeexplore.ieee.org/document/9593947
6. LR6: https://ieeexplore.ieee.org/document/10242512
7. LR7: https://ieeexplore.ieee.org/document/10698061
8. LR8: https://ieeexplore.ieee.org/document/6717252
9. LR9: https://ieeexplore.ieee.org/document/9593984
10. LR10: https://ieeexplore.ieee.org/document/10252030
11. LR11: https://ieeexplore.ieee.org/document/8645984
12. LR12: https://www.swervedrivespecialties.com/
13. LR13: https://docs.wpilib.org/en/stable/docs/software/kinematics-and-odometry/swerve-drive-kinematics.html
14. LR14: https://www.turtlebot.com/
15. LR15: https://nav2.org/
16. LR16: https://github.com/SteveMacenski/slam_toolbox
17. LR17: https://github.com/ros-navigation/navigation2/blob/humble/nav2_voxel_grid/README.md
18. R1: https://wiki.ros.org/sw_urdf_exporter (ROS1 URDF Exporter)
19. R2: https://www2.dmst.aueb.gr/dds/pubs/inbook/beautiful_code/html/Spi07g.html#:~:text=All%20problems%20in%20computer%20science,envisioned%20the%20modern%20personal%20computer. (Butler Lampson Quote)
20. R3: https://github.com/ros-controls/ros2_controllers
21. R4: https://ieeexplore.ieee.org/document/6147128
22. https://www.outlookbusiness.com/start-up/news/meet-blue-the-cute-little-ai-robot-built-by-nvidia-disney-and-google
23. https://www.sciencedirect.com/science/article/pii/S0263224119302489?casa_token=6tl98m6DxOQAAAAA:HaJ0SL0x3tIQ6LKtYFuq_PTn2jUMhKIraj6jPxuWhBSjUVvQ-S2s9FfCJhsnK6Xoxz9aRbJn
24. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/
25. https://foxglove.dev/
26. https://www.oemoffhighway.com/engineering-manufacturing/press-release/21940684/daimler-trucks-north-america-llc-daimler-trucks-develops-autonomous-truck-platform-for-sae-level-4-autonomous-vehicles
27. https://www.forsythcreations.com/swerve_drive
28. https://datasheets.raspberrypi.com/cm5/cm5-datasheet.pdf