**ISTANBUL TECHNICAL UNIVERSITY**
Department of Electrical and Electronics Engineering
Control and Automation Engineering

# *FINAL PROJECT REPORT*

**DATE :** 30/12/2025

**PREPARED BY**

**NAMES AND NUMBERS:**

BERKE ALGÜL 040230762

HAMDI SELIM EKŞI 040220551

CEM ŞENGÜL 040210572

ABDULKADIR OKTAY UĞURLU 040210522

YILMAZ EKMEKÇI 040200567

TUNÇ BORA ELÇIN 040190639

**COURSE CODE:** KON 440E

**COURSE TİTTLE:** LEGGED ROBOTICS

**CRN:** 15345

**LECTURER:** DR. SEMIH BEYÇIMEN

I confirm that the work presented in this report is my own. Where information has been derived from other sources, I confirm that this has been indicated in the report. I accept all legal responsibilities for plagiarism.

## *CONTENTS:*

## 1. INTRODUCTION

This project details the development and implementation of a control system for a quadrupedal robot utilizing the CHAMP (Open-Source Quadruped Robot Control Development Framework). The primary objective is to design and validate a robust algorithm for stable and dynamic locomotion across various terrains, showcasing fundamental skills in legged robotics control and state estimation. The scope includes configuring the robot's kinematic and dynamic models, developing a custom or optimized gait pattern generation routine, and integrating the control system with a high-fidelity simulation environment. This report will cover the project's methodology, the architecture of the designed control system, and a comprehensive analysis of the robot's performance metrics through both simulation and, potentially, hardware implementation.

## 2. ROBOT MODEL AND TECHNICAL FRAMEWORK

The software infrastructure is built upon ROS 2 Humble Hawksbill, chosen for its long-term support (LTS) and performance enhancements crucial for real-time control and distributed systems like the quadruped's control stack. The core control is managed by the CHAMP (Open-Source Quadruped Robot Control Development Framework), which primarily handles the high-level tasks of motion planning and gait generation. This framework models a generalized quadruped robot, typically a 12-Degrees-of-Freedom (DOF) system similar to platforms like the MIT Mini Cheetah or SpotMicroAI, which allows for a high degree of kinematic agility. Within the ROS 2 environment, CHAMP leverages several key components: the controller_manager node manages the interface between the high-level commands (e.g., walk, trot, gallop) and the joint-level controllers; the robot_state_publisher continuously broadcasts the robot's current pose and joint transformations using the Unified Robot Description Format (URDF) model; and the ros2_control framework provides a generic interface to the robot's hardware.

On the hardware front, the robot's control relies on an integrated sensor suite: an Inertial Measurement Unit (IMU) is indispensable for precise state estimation and maintaining balance, providing critical feedback on the robot's angular velocity and linear acceleration. Additionally, high-resolution joint encoders on all 12 motors supply the necessary position feedback to the low-level joint-space controllers, and for autonomous functionality, a perception sensor such as a LiDAR or depth camera is included to facilitate environment mapping (SLAM), obstacle avoidance, and path planning using the ROS 2 Navigation stack.

*Figure 1: Cham Quadrupeds*

## 3. REPOSITORY OVERVIEW AND OPERATION

### a) Overview of the Repo Structure

The CHAMP repository (the name of the repository provided to us) consists of multiple modular packages, conforming to the ROS package structure. This modular structure facilitates system maintenance and expansion. A brief overview of the main packages offered within the repository is as follows:

- **champ (package 1)** This is the core package containing the basic gait, kinematics, and control algorithms for quadruped robots.
- **champ_base (package 2)** It houses the ROS nodes that manage the robot's basic state information and communicate with the control layers.
- **champ_bringup (package 3)** It contains the necessary launch files for starting the system. It allows you to start all the robot's components with a single command.
- **champ_config (package 4)** The robot contains configuration files that include geometry, gait parameters, and control settings.
- **champ_description (package 5)** It contains URDF files that define the physical model of the robot. Visualization and simulation are performed through this package.
- **champ_gazebo (package 6)** Gazebo enables integration between the simulation environment and CHAMP.

- **champ_msgs (package 7)** It includes defined custom ROS message types for control and status information.
- **champ_navigation (package 8)** It enables autonomous navigation of the robot by providing integration with the ROS2 Navigation Stack (Nav2).

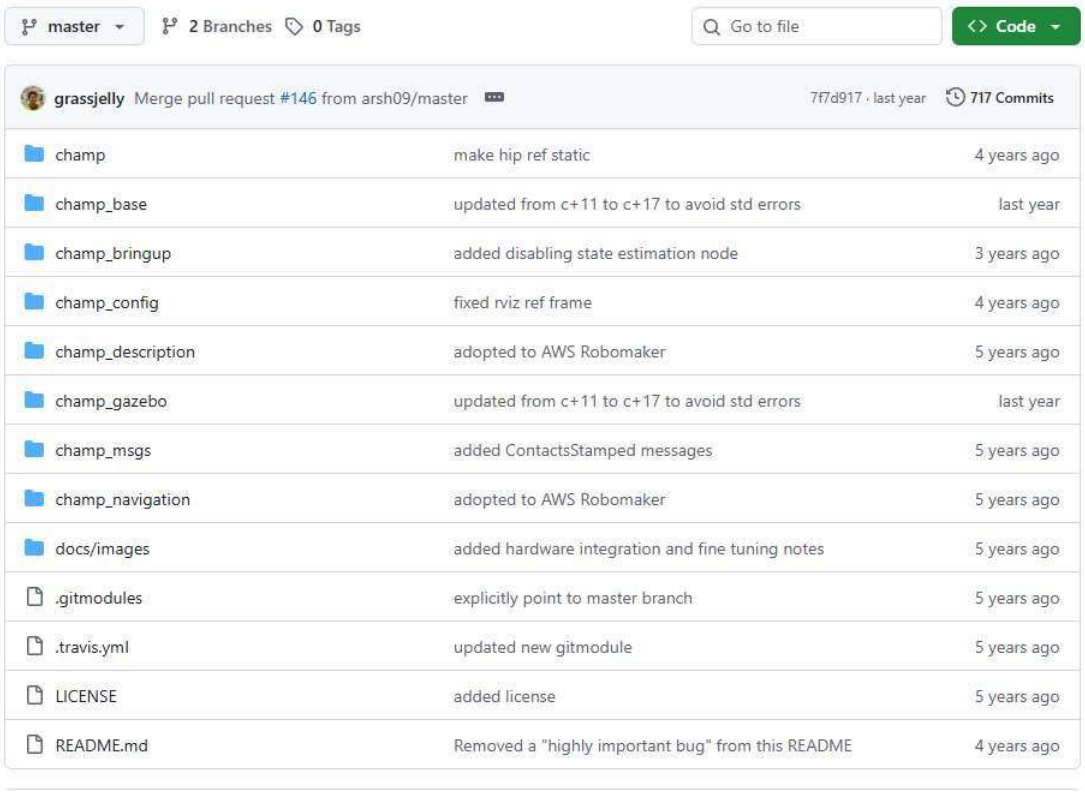


*Figure 2: Repo Structure*

## *b) The Installation of Repo*

This section describes the step-by-step and detailed installation of the CHAMP (ROS2) repository in a ROS2 runtime environment. First, ROS2 (e.g., ROS2 Foxy or Humble) will be installed on the system. Next, a workspace will be created. After that, the repository will be cloned and dependencies will be installed. Finally, the compilation process will be performed.

### *i. Preparing the ROS2 Environment*

First, ROS2 (such as ROS2 Foxy or Humble) must be installed on the system. Then, the 'rosdep' tool for dependency management is installed and updated. This step ensures that the ROS packages used by CHAMP are automatically installed on the system.

The code that needs to be entered into the terminal:

```
sudo apt install -y python3-rosdep
rosdep update
```

*Figure 3: This is the code that needs to be entered into the terminal for preparing the ROS2 environment section.*

### *ii. Creating a Workspace*

ROS2 projects are typically run within a colcon workspace. A workspace is created using the following commands:

```
mkdir -p ~/champ_ws/src
cd ~/champ_ws/src
```

*Figure 4: This is the code that needs to be entered into the terminal for Workspace section.*

### *iii. Cloning the Repository*

The CHAMP repository must be cloned recursively along with the ROS2 branch. This is because the repository contains submodules. Additionally, the extra package used for teleoperation must be cloned separately. This package allows sending movement commands to the robot via keyboard or joystick.

```
git clone --recursive https://github.com/chvmp/champ -b ros2
```

```
git clone https://github.com/chvmp/champ_teleop -b ros2
```

*Figure 5: This is the code that needs to be entered into the terminal for Cloning section.*

### *iv. Establishing Addictions*

Return to the workspace root directory and install all ROS dependencies required by CHAMP using rosdep. This command scans the packages in the src folder, identifies missing ROS dependencies, and performs an automatic installation via the system package manager.

The code that needs to be entered into the terminal:

```
cd ~/champ_ws
rosdep install --from-paths src --ignore-src -r -y
```

*Figure 6: This is the code that needs to be entered into the terminal for Addictions section.*

### *v. Compilation Process*

Once all dependencies are established, the project is compiled, and after compilation is complete, environment variables need to be loaded.

```
colcon build
source install/setup.bash
```

*Figure 7: This is the code that needs to be entered into the terminal for Compilation section.*

## *c) Running the Repository*

### *i Running on RVIZ*

The following command is used to view the robot's kinematic structure and walking movements on RVIZ. This command initializes the necessary control nodes, loads the robot's URDF model, and visualizes the robot's status by opening the RVIZ interface.

```
ros2 launch champ_config bringup.launch.py rviz:=true
```

*Figure 8: This is the code that needs to be entered into the terminal for RVIZ section.*

### *ii Teleoperation and Manual Control*

The teleoperation package can be activated to manually control the robot using a keyboard or joystick.

```
ros2 launch champ_teleop teleop.launch.py
```

*Figure 9: This is the code that needs to be entered into the terminal for Teleoperation section.*

### iii. *Gazebo Simulation*

Gazebo is launched to test the robot's physical behavior in a simulation environment.

```
ros2 launch champ_config gazebo.launch.py
```

*Figure 10: This is the code that needs to be entered into the terminal for Gazebo section.*

### iv. *Using Navigation (Nav2)*

CHAMP can work in conjunction with the ROS2 Navigation Stack. When the navigation stack is activated, the robot attempts to reach the target by sensing its environment and planning its movements.

## 4. *Mapping Process*

### *a) Architectural Design and Components*

The mapping architecture in CHAMP is SLAM-driven and adheres to a modular ROS 2 design philosophy. The system processes sensor input from a 2D LiDAR (LaserScan messages) via the core **slam_toolbox** engine, utilizing TF2 to manage coordinates from the map frame down to the base link. The resulting output is a 2D occupancy grid that serves downstream consumers such as the Nav2 Map Server, AMCL, and global planners. While the mapping system operates independently of navigation, it is designed to hand off the generated map to Nav2 for localization and planning.

Functionality is distributed across multiple packages to maintain modularity. **champ_navigation** provides the launch files that coordinate integration, while **champ_config** houses SLAM parameter YAML files and robot-specific configurations. These internal components work alongside external dependencies like **slam_toolbox** and **nav2_map_server**. This separation ensures that mapping logic remains reusable and configurable without requiring modifications to core control code.

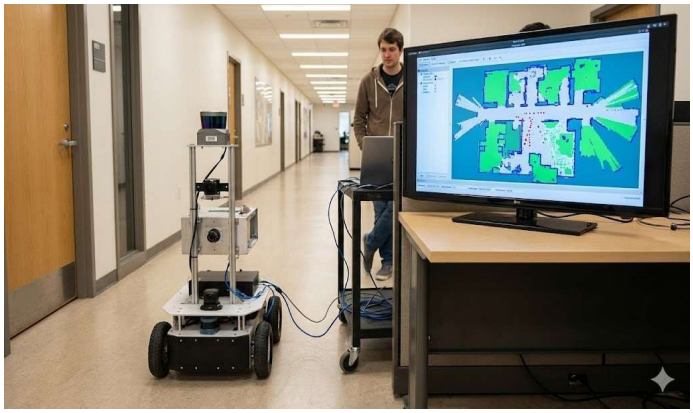

*Figure 11: The picture of Architectural Design*

### *b) Launch Configuration and Dynamic Parameters*

The **slam.launch.py** file acts as the primary entry point for the mapping process, serving as a wrapper that configures and includes SLAM Toolbox launch descriptions rather than instantiating nodes directly. Logic is encapsulated within the standard **generate_launch_description()** function, ensuring compatibility with ROS 2's declarative system.

To facilitate flexibility, mapping configuration relies on runtime-resolved launch arguments. Parameter file paths are declared via **LaunchConfiguration** and resolved at execution time rather than being hard-coded. This design supports rapid experimentation with tuning files, seamless switching between simulation and hardware, and reproducibility across deployments. The SLAM process specifically includes **online_sync_launch.py** from **slam_toolbox**, enabling synchronous operation, real-time scan matching, and continuous map updates while the robot moves.

### *c) The LiDAR Mapping Pipeline*

The mapping pipeline enforces a strict data flow where the LiDAR publishes sensor_msgs/LaserScan data, and TF provides the necessary transforms between base_link, odom, and map. The SLAM Toolbox aligns these incoming scans using scan matching to incrementally update the global occupancy grid. Accurate TF alignment is critical, as errors in odometry or frame transforms will directly degrade map quality. The system assumes a locally planar environment and a 2D LiDAR perspective with limited vertical variation.

Consequently, slopes and uneven terrain can distort measurements and negatively impact scan alignment, leading to map deformation.

### *d) Integration with Navigation and Execution*

Mapping and navigation operate as decoupled but cooperative subsystems. During the mapping phase, slam_toolbox publishes the map, whereas during navigation, Nav2's Map Server consumes it, AMCL uses it for probabilistic localization, and planners compute paths relative to it. This separation allows the system to perform mapping first and then switch to a localization-only mode, reusing the same map across multiple runs while maintaining deterministic behavior.

When a user executes the command ros2 launch champ_config slam.launch.py, the system follows a specific sequence: the launch description is parsed, file paths are resolved, runtime arguments are substituted, the nodes are composed into the graph, and real-time SLAM begins execution. This workflow parallels the navigation startup sequence, reinforcing architectural symmetry.

### *e) Simulation, Terrain, and Limitations*

Mapping is primarily tested in Gazebo, which allows for the alteration of spawn conditions, physics parameters, and terrain profiles. However, the presentation identifies terrain-induced challenges, such as reduced traversability impacting odometry accuracy and slopes violating 2D SLAM assumptions. Additionally, perspective distortion can affect scan matching, potentially requiring alternative routes to maintain map consistency.

Key limitations of the current approach include its dependence on accurate odometry, sensitivity to non-planar terrain, and the lack of native 3D environment representation or terrain-aware SLAM correction. Despite these constraints, the system remains effective for indoor environments, structured outdoor settings, and navigation-focused research.

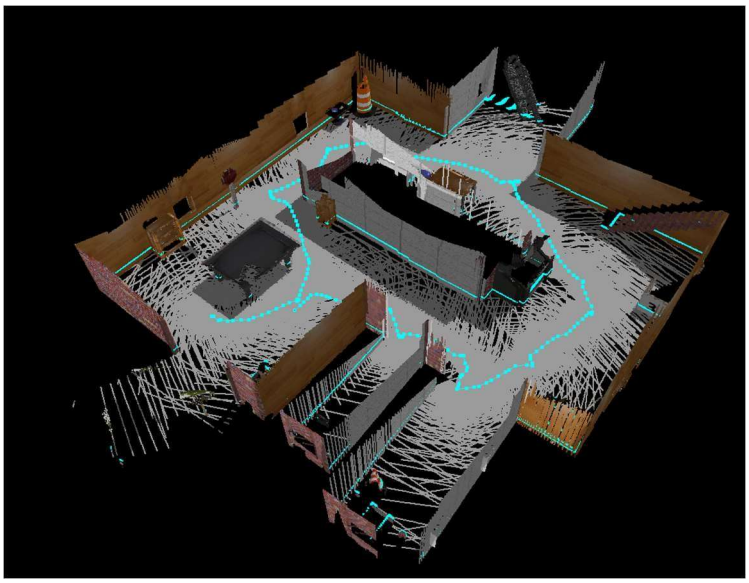

*Figure 12: 3D Mapping Picture*

## 5. Navigation

### a) Analysis of Maps.launch.py

The Maps.launch.py script serves as the primary entry point for initializing the Navigation2 stack. It is structured as a declarative Python script that constructs a LaunchDescription object, encapsulating the entire logic within the standard generate_launch_description function. The script begins by importing essential modules from the launch and launch_ros libraries. This includes os for operating system-level path manipulation and ament_index_python.packages—specifically the get_package_share_directory function—which dynamically locates the installation paths of ROS 2 packages like champ_navigation and nav2_bringup to ensure portability across different file systems. Additionally, launch.substitutions.LaunchConfiguration is used to enable lazy evaluation of variables, allowing users to pass arguments via the command line at runtime.

In terms of path resolution, the code defines absolute paths for configuration files. champ_navigation_dir locates the root of the current package, while launch_dir points to the standard Navigation2 bringup directory, allowing the script to inherit pre-built navigation behaviors. To maintain flexibility without code modification, the script uses DeclareLaunchArgument actions to define the input interface. Key arguments include map, which specifies the file path for the occupancy grid map (defaulting to maps/map.yaml), params_file for loading tuning parameters for Costmaps and Planners, and use_sim_time to toggle between the system clock and the simulation clock.

### b) Integration and Process Inclusion

The core logic of the navigation script utilizes the IncludeLaunchDescription action. Instead of rewriting navigation initialization code, it wraps the standard bringup_launch.py provided by the Nav2 stack using PythonLaunchDescriptionSource. The launch_arguments dictionary maps local variables to the arguments expected by the included file. This mechanism passes Champ-specific map and configuration files into the generic Nav2 system, effectively customizing the generic stack for this specific robot.

### c) Analysis of slam.launch.py

The slam.launch.py script is responsible for the synchronous localization and mapping process. Structurally similar to the navigation script, it uses the generate_launch_description function to return the system state description. The script handles dynamic parameter loading by identifying the location of the SLAM configuration file via a substitution variable (slam_params_file), which resolves when the user executes the launch command.

For node instantiation, the script executes the SLAM algorithm by including the online_sync_launch.py file from the slam_toolbox package. It explicitly passes the slam_params_file to the toolbox, which contains algorithmic settings such as scan matching thresholds and map update intervals that determine how LIDAR data is converted into a map.

### d) Execution Workflow

When a user executes the command **ros2 launch champ_navigation navigate.launch.py**, the Python interpreter performs a specific sequence of operations. First, it **Parses** the **generate_launch_description** function. Next, it **Resolves** absolute paths for maps and configurations using **os.path** and **ament_index.** It then **Substitutes LaunchConfiguration** placeholders with values provided in the terminal. Following this, it **Composes** a process graph that links **champ_navigation** inputs to nav2_bringup nodes. Finally, it **Executes** the process, spinning up the Map Server, AMCL, and Controller Server nodes with the specific parameters defined in the script.

## 6. Results

### a) System Validation and Locomotion Outcomes

The project resulted in a successful system bring-up, where the quadruped robot model was correctly configured using URDF, allowing for the accurate visualization of the kinematic chain, joint states, and TF frames in RViz. All core ROS 2 nodes required for control and visualization were launched successfully. Regarding locomotion, the robot demonstrated stable walking behavior under teleoperation using CHAMP's built-in gait generation mechanisms. The system smoothly executed forward, backward, lateral, and rotational commands, confirming the correct operation of the hierarchical control structure.

Physical behavior was further validated in the Gazebo simulation environment, where tests on foot-ground contact, swing–stance transitions, and body stability provided confidence in the correctness of the dynamic configurations. Additionally, the project established autonomous navigation capabilities; the integration with slam_toolbox enabled online mapping while Nav2 provided global and local path planning. The launch files Maps.launch.py and slam.launch.py successfully linked CHAMP-specific configurations with the generic navigation stack, allowing the robot to navigate toward user-defined goals in simulation.

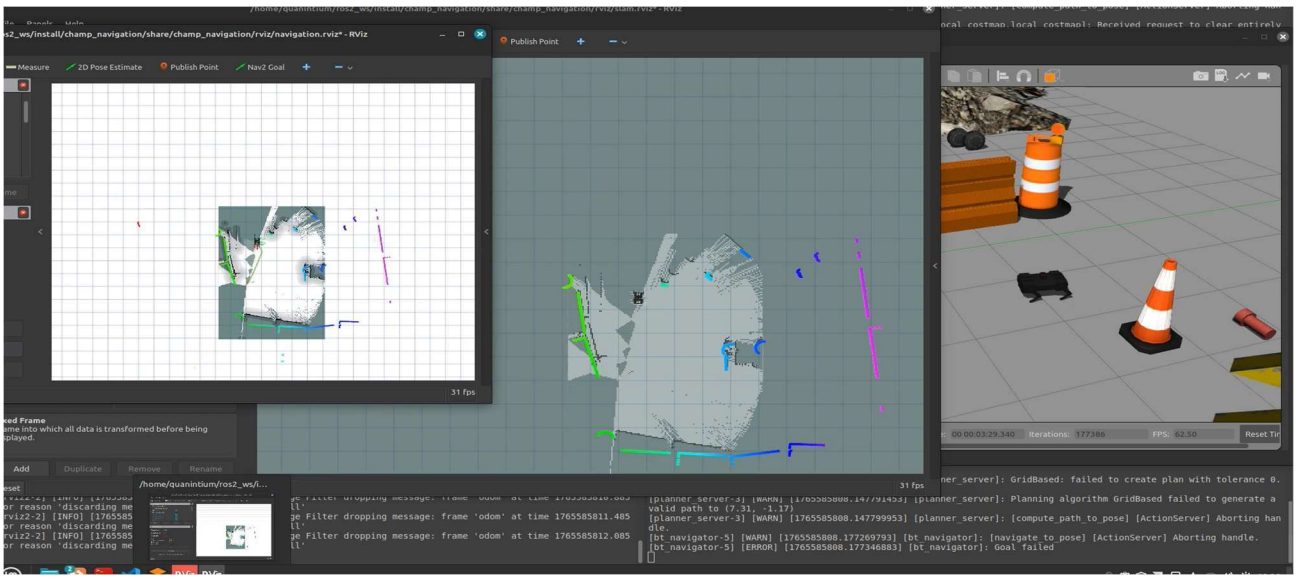


*Figure 13: Picture from Gazebo and RVİZ*

## b) Performance Evaluation

System performance was evaluated qualitatively through simulation-based experiments focusing on stability, command tracking, foot contact consistency, and navigation behavior. Specifically, the evaluation assessed the robot's ability to maintain balance without excessive oscillation, its responsiveness to velocity commands, and the reliability of stance phases with minimal slipping. Navigation was judged on successful map generation and goal tracking. Overall, the results confirm that RViz visualization, Gazebo locomotion, teleoperation, and Nav2 integration are functional, aligning with the current development status of the CHAMP ROS 2 port.

## c) Challenges and Future Work

Despite the project's successes, several challenges and limitations were identified that define the roadmap for future development. The simulation did not give proper results in a not flat terrain. The sensors could not map the area properly when the surface is not flat. A key difficulty is parameter sensitivity, where gait metrics such as stance duration, swing height, and walking height significantly affect stability and require careful tuning. Additionally, a simulation–reality gap exists; accurate modeling of friction, mass, and inertia is essential, as mismatches can lead to unrealistic behaviors in simulation. The maturity of the ROS 2 port also remains a factor, with components like velocity smoothing and full code refactoring still under development, which limits immediate deployment on physical platforms. To address these issues, future work will focus on gait optimization to automate tuning for different terrains and enhanced state estimation using techniques like Extended Kalman Filters (EKF) to fuse IMU and joint encoder data. The project also aims to move toward hardware implementation by integrating ros2_control, motor drivers, and real sensor interfaces. Finally, future evaluations will incorporate quantitative performance metrics, logging data such as body roll/pitch RMS, center-of-mass oscillations, and velocity tracking errors to provide a more rigorous analysis of system performance.

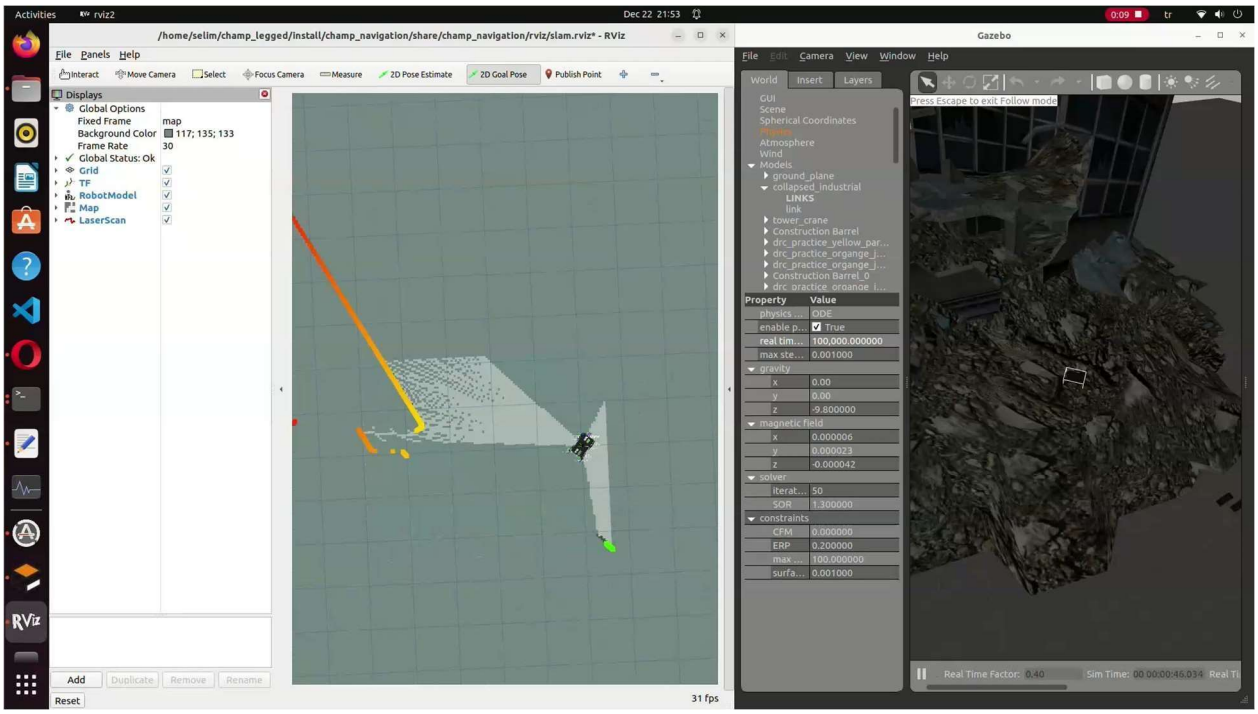


*Figure 14: Simulation Results From Different Terrain*

## 7. Conclusion

In conclusion, this project successfully demonstrates that the CHAMP framework, integrated with ROS 2 Humble, provides a viable, robust, and flexible foundation for developing quadrupedal robot locomotion and autonomous navigation systems in simulation. The implemented system proved capable of achieving stable, repeatable walking behavior and executing goal directed motion through a hierarchical control structure. By effectively linking CHAMP specific configurations with the generic ROS 2 Navigation Stack (Nav2) and slam_toolbox, the project validated that complex autonomous behaviors such as online mapping and global path planning can be reliably executed in high fidelity environments like Gazebo. While the transition to real-world hardware requires further refinement to address dynamic modeling and parameter tuning challenges, the work establishes a critical baseline for future research, confirming that the current architecture is functionally complete for simulation-based research and prototyping.