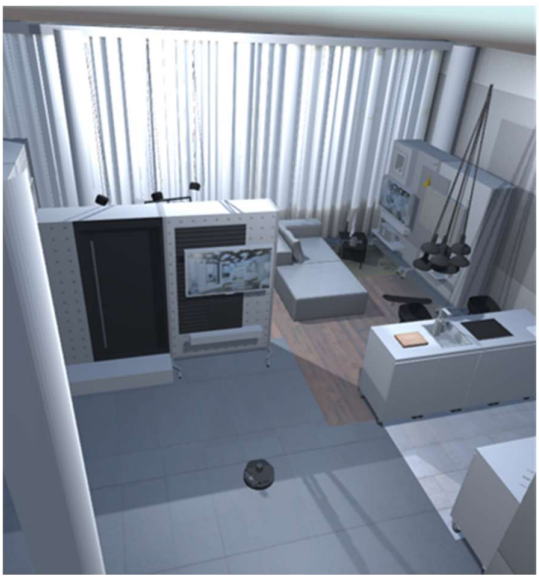
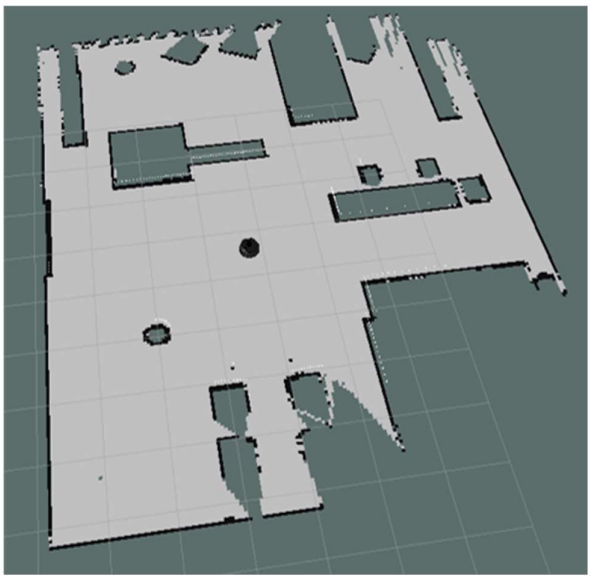
DEGREE PROJECT FOR MASTER OF SCIENCE WITH SPECIALIZATION IN ROBOTICS AND AUTOMATION
DEPARTMENT OF ENGINEERING SCIENCE
UNIVERSITY WEST

Simulation of Mobile Robots with Unity and ROS

- A Case-Study and a Comparison with Gazebo

Anna Konrad





UNIVERSITY WEST

A THESIS SUBMITTED TO THE DEPARTMENT OF ENGINEERING SCIENCE

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE WITH SPECIALIZATION IN ROBOTICS AND AUTOMATION

AT UNIVERSITY WEST

2019

Date:	June 12, 2019
Author:	Anna Konrad
Examiner:	Bo Svensson
Advisor:	Alexander Sung and Kati Radkhah-Lens, Robert Bosch GmbH Anders Appelgren, Högskolan Väst
Programme:	Master Programme in Robotics and Automation
Main field of study:	Automation with a specialization in industrial robotics
Credits:	120 Higher Education credits (see the course syllabus)
Keywords:	Simulation, Mobile robots, Unity, ROS, Gazebo, Physics engine
Template:	University West, IV-Master 2.9
Publisher:	University West, Department of Engineering Science S-461 86 Trollhättan, SWEDEN Phone: + 46 520 22 30 00 Fax: + 46 520 22 32 99 Web: www.hv.se

Summary

Simulation software is becoming an increasingly important tool for automation systems both in industry and research. Within the field of mobile robotics, it is used to evaluate the performance of robots regarding localization, motion planning and control. In order to evaluate the performance of a mobile robot based on a simulation, the simulation must be of sufficient precision and accuracy. A commonly used middleware in robotics with increasing usage and importance is ROS (Robot Operating System). The community and the amount of robots in industry and research that are using ROS is growing. In the ROS community, the built-in simulator Gazebo is most often used for simulating purposes. Apart from Gazebo, popular game engines, like Unity, shift more in the focus of researchers for simulating robots. Those engines provide high capabilities in customized visualisations of simulations, the user interface and for example additional plugins for machine learning. Simultaneously, the physics engines which calculate the physical behaviour of objects in regard to their environment in those game engines are improving their performance. The performance of Unity as a simulator in the ROS environment should be tested in this thesis work. While testing the performance of the simulator, its ability to adapt to different scenarios and applications should be rated. For that purpose, benchmarks were created and the results were compared to reality and Gazebo.

The simulations in Unity could depict the basic behaviour of mobile robots for varying circumstances. The simulation for the specific setup in Unity was more detailed and closer to reality than the simulation in Gazebo. A major influence for that result was a non-existing controller for the wheels in Gazebo. Quantitative benchmarks showed a similar behaviour to reality in the Unity simulations. The results were highly dependent on the friction values and other simulation parameters. With the chosen setup, Unity showed systematic and non-systematic errors in the end-position of predefined paths. A SLAM case study presented the possibility to use Unity in combination with SLAM algorithms on ROS. It was possible to implement other algorithms and simulate required sensors for data acquisition in Unity.

Overall, Unity was rated to be suitable for the simulation of mobile robots in robotics research if no accurate simulation of the robots' properties is needed. Of major importance when conducting simulations with mobile robots in Unity are the used robot model and the chosen parameters in the simulation. Only if the robot model is validated, the results can be judged accordingly. To reduce the effect of the parameter settings in Unity on the results, no setups to solely examine the robot's hardware should be chosen. Instead, the simulations can be used for testing, improving and evaluating control algorithms on the robot. A huge opportunity is seen in the usage of automatically and synthetically generated simulation data for teaching deep learning algorithms.

Preface

This thesis work was created in cooperation with the Robert Bosch GmbH during the master program "Robotics and Automation" at University West. I would like to thank my supervisors at the Robert Bosch GmbH, Dr. Alexander Sung and Dr. Kati Radkhah-Lens, for guiding me through this thesis, helping me with my measurements and sharing their knowledge with me.

For enabling me to do this master thesis at Bosch and many good words of advice, I would like to thank Dr. Ralph Lange from the Robert Bosch GmbH. Special thanks go to Musa Morena Marcusso Manhaes from the Robert Bosch GmbH for conducting the measurements in Gazebo.

I also want to thank my supervisor, Anders Appelgren, and examiner, Dr. Bo Svensson, from University West for their support during the whole time of my studies at University West. Thank you for your dedication for your students and all the work you put into teaching us. Last but not least I would like to thank my family, which encouraged and supported me all my life. Without you, I would never got to where I am today.

Affirmation

This master degree report, *Simulation of Mobile Robots with Unity and ROS*, was written as part of the master degree work needed to obtain a Master of Science with specialization in Robotics and Automation degree at University West. All material in this report, that is not my own, is clearly identified and used in an appropriate and correct way. The main part of the work included in this degree project has not previously been published or used for obtaining another degree.

Anna Konrad 12.06.2019

Signature by the author Date

Anna Konrad

Contents

Preface

SUMMARY III

PREFACE IV

AFFIRMATION V

CONTENTS VI

Main Chapters

1 INTRODUCTION 1

1.1 LIMITATIONS 2
1.2 AIM 2

2 RELATED WORK 4

2.1 ROS 5
2.2 UNITY 7
2.3 MAIN CHARACTERISTICS IN PHYSICS ENGINES 9
2.4 RESEARCH IN PHYSICS ENGINES 10
2.5 EXPERIMENTS FOR PHYSICS ENGINE EVALUATIONS 12

3 METHOD 14

4 SIMULATION SETUP AND SCENARIOS 15

4.1 UNITY SETTINGS 15
4.2 ROSSHARP 17
4.3 ROBOT MODEL 19
4.4 TEST SCENARIOS 20

5 RESULTS AND DISCUSSION 28

5.1 STACKING TEST 28
5.2 COLLISION TEST 30
5.3 INTEGRATOR TEST 30
5.4 μ-SPLIT 31
5.5 RAMP 34
5.6 FRICTION TURN 37
5.7 BASIC DRIVE 39
5.8 UMBMARK 40
5.9 SLAM CASE STUDY 45

6 CONCLUSION 50

6.1 FUTURE WORK AND RESEARCH 51
6.2 CRITICAL DISCUSSION 52
6.3 GENERALIZATION OF THE RESULT 52

7 REFERENCES 53

Appendices

A. SOFTWARE AND HARDWARE SETUP

B. ROBOT MODEL

C. C# SCRIPT FOR WHEEL-MOTOR IN UNITY

1 Introduction

In recent years, mobile robots are becoming more and more common in industry and private households. According to the definition of the International Organization for Standardization, a robot is an "actuated mechanism programmable in two or more axes with a degree of autonomy, moving within its environment, to perform intended tasks" [1]. A mobile robot is defined as such a robot which is "able to travel under its own control" [1]. In industry, self-driving vehicles are used in factories to transport material or machines between processes efficiently. In the sector of service robotics in private households, mobile robots are most commonly used as vacuum cleaners, lawnmowers or floor and window cleaners. They assist the customers to clean their house and garden efficiently.

When developing those mobile robots or planning their usage, simulation can be used in various stages. Simulation can be a huge advantage when real robot prototypes or products are not available or cannot be used due to other circumstances. During the development, simulation can be used to assess the basic hardware functionality. In addition to that, the algorithms for localization, motion planning or control can be tested, improved and integrated continuously. Simulation software can be of great advantage not only if no real hardware is available, but also if the scenario is difficult to test in reality or if the quantity of required experiments/iterations is too large to be efficiently tested in reality. In those cases the simulation can be used to judge the performance of the robot and/or its concept. This usage can increase the efficiency and decrease the costs of the development.

Apart from all the advantages, the simulation will always stay a simplified depiction of the reality. Assumptions have to be made when simulating real environments and thus decreasing the accuracy of the simulation. The simulation can only be used in the previously mentioned setups if the simulation is accurate enough to let the user draw conclusions out of the results. If the measuring uncertainty is greater than the accuracy that is needed for the evaluation, using the simulation is not applicable.

Unity [2] is a popular game-engine and widely used for the development of video games. It comes with a high-functional and user-friendly graphical user interface. It uses PhysX by NVIDIA [3] as physics engine to simulate the behaviour of objects in regard to their physical environment and forces acting upon them. Unity's user-friendly interface, its flexibility and its functions in the area of artificial intelligence [4] and human-machine-interaction [5] are beneficial for the choice of a mobile robot simulator. It can be extended with toolkits that enable the development of learning environments with a Python API in Unity [4]. Complex sensors and physical environments can be modelled and dynamic multi-agent interaction is supported [6].

The usage of Unity as a robotic simulator with learning environments could be useful for training learning algorithms. Data can easily be produced by automatically creating environments and acquiring data through simulations. This data can then be used for training the learning algorithms. Usually, a huge amount of data is needed for training a learning algorithm in a sufficient way. An application example in the field of mobile robotics is machine learning in combination with SLAM (Simultaneous Localisation and Mapping) algorithms of mobile robots. There have already been attempts to

extend and improve SLAM with deep learning algorithms [7]. The main goals for implementing deep learning methods into SLAM algorithms are to improve the efficiency and robustness of those algorithms.

For teaching learning algorithms, an enormous amount of data is needed. It is likely that the more data from different environments can be used for training, the more accurate and robust the results from the learning algorithms will become. It could be a major advantage if simulation software, like Unity and Gazebo [8], can be used for collecting data and teaching algorithms. When the same data can be achieved with simulation software, there would be no need to perform expensive experiments with a real robot. It could be faster and more efficient than the conventional methods. There would be the possibility to create the simulation environments automatically and take away a lot of work and effort from the researchers and developers [9].

Gazebo is a popular simulator used in the ROS [10] (Robot Operating System) community. It is published under an open-source Apache 2.0 license. The user can switch between ODE, DART, Simbody or Bullet as physics engine when starting Gazebo. As Gazebo is the primary simulation tool of ROS users and ROS is used for an increasing amount of robots [11], it is already used for simulating robots and specifically mobile robots.

Regardless of the usage of both simulators in robotics, the accuracy and performance of these simulators in regard to reality still has to be evaluated. It is of essential importance to judge the limitations and performance of those simulators in order to use simulation results. Only if the deviation to reality is known, the results can be used within reason. To assess the limitations and performance of simulators is a challenging and ongoing task. It is highly dependent on the settings of the simulators as well as the used physics engines and their implementation into the simulators. The simulators and some of the physics engines are still under ongoing, intensive development, which makes the evaluation even more challenging.

The thesis is structured as follows: in the 2nd chapter, the literature study with related work and background is presented. The 3rd chapter describes the method which was used for the work in this thesis. Chapter 4 describes the work which was done in the experimental phase, including the settings in Unity, the setup of the environment, the used model of the TurtleBot2 robot and the chosen test scenarios. In the 5th chapter, the results of the experiments are shown and discussed. The 6th and last chapter concludes and generalizes the work, as well as describes possible future work.

1.1 Limitations

The limitations of this thesis are as follows. The simulations and experiments in Gazebo are not conducted by the author of this thesis, but by employees of the Robert Bosch GmbH. The accuracy, limitations and performance of Unity is only judged by the performance of nine different benchmarks. The comparison of Unity and Gazebo is only based on four of those benchmarks. The benchmarks are only tested with one type of mobile robot, namely a TurtleBot2. No other robot models and robot types are used.

1.2 Aim

This thesis aims to examine the simulation of mobile robots with Unity and ROS. The simulation results should be compared with reality and Gazebo in regard to the performance of mobile robot simulations. Within the performance of the simulators, their

adaptability for different scenarios and the possibility for implementing different algorithms and methods should be tested. For evaluating Unity and comparing it to reality and Gazebo, systematic test scenarios should be determined and evaluation parameters for the performance of the simulation need to be identified. Then, Unity should be compared to reality and Gazebo for mobile robot simulation with reproducible and reasonable evaluation parameters.

2 Related work

Unity is a real-time simulation platform, whose original focus lies on the development and creation of video games. Despite its background as a game engine, Unity has already been used as a platform for robotic simulations. Some of the research projects that used Unity for robotic simulations are presented in the following.

With a platform called "The Robot Engine", Bartneck, Soucy, Fleuret and Sandoval [5] created a framework to use Unity for simulating and investigating human-robot interaction. In this approach, Arduino-based robots were integrated into and controlled by Unity. Speech recognition as well as movement recognition was used for the human-robot interaction. It was stated that Unity was a desirable tool for this purpose, as it allowed the developer to easily animate humans and robots. The capability of human animation in Unity was also used in a research regarding mixed reality for robotics [12]. Next to Unity, also Gazebo and V-Rep were used as simulators for unmanned aerial vehicles (UAVs) in different environments. The researchers stated that a combination of several robotic simulators in different testing phases could enhance the accuracy in testing of robots, as all of them had different specialities and strengths.

Unity has been linked with ROS by Hussein, García and Olaverri-Monreal [13] for simulating and controlling small mobile research platforms and, in a further step, cars as autonomous vehicles. Unity's developer environment was only available for Windows and Mac OS. ROS was only available for Linux. For simulating and controlling intelligent vehicles, Unity was connected to ROS by using the so-called "rosbridge"-module [14]. In that way, similar environments could be simulated in Gazebo and Unity. Both simulations could subscribe to the same control messages and publish data like odometry information. Even though the basic performance and comparability of such simulations was given in this research, the focus did not lie on the accuracy or precision on either of the simulators. No comparison in terms of navigation, mechanics or sensors was done, neither in regard to Gazebo, nor in regard to the real world.

The rosbridge has also been used to connect Unity with ROS for VR (virtual reality) in a research by Mizuchi and Inamura [15]. This approach aimed to improve the reusability of robot-control software and decrease the costs for developing VR interface modules. With the usage of ROS, the first aim was addressed, as its structure enhances the reusability of robotic software. While this approach also used the rosbridge module for binding ROS to Unity, like in the previous research [13], it used a slightly different approach while doing so. Instead of only using JSON data, which was a text-based format to exchange data, a similar format called BSON was employed in parallel. The usage of BSON had the advantage to speed up the communication process. This was seen especially useful by the authors for the exchange of large sets of data, like images, in real-time. As the target application in this research was VR, the fast communication of visual information as images was especially desirable.

A third research that linked Unity with ROS was conducted by Codd-Downey, Forooshani, Speers, Wang and Jenkin [16]. The authors also used the rosbridge package to realise the connection between Unity and ROS with a special focus on VR applications. A special focus in this research lay on the different architectures of ROS and VR systems. While the architecture was based on message passing, the VR systems were

typically working in a rendering loop. With the rosbridge, those different architectures could be connected and successfully used for simple VR experiments.

Unity and ROS should be connected for this thesis work to ensure the usage of the same robot control software for the robot in reality and the simulated robots in Unity and Gazebo. To get more insight on the structure of ROS, what it was used for and why it could be important for mobile robot simulation, the following section describes ROS in more detail.

2.1 ROS

ROS (Robot Operating System) is – despite its name – not an ordinary operating system but a middleware working on top of a host operating system and providing services similar to those of an operating system. It is designed to be free, open-source, multilingual, thin, peer-to-peer and tools-based. It has been developed during projects at Stanford University and Willow Garage and is nowadays maintained by Open Robotics [17]. One of the main goals of ROS is to reuse and share code in robotics research and development. In that way researchers could collaborate with each other globally on the same framework by using and extending code of other researchers to solve common problems. The ROS framework can be used with Python, C++ and Lisp by default. With additional implementations, any modern programming language can be used, which makes it suitable for diverse usage. For programming in ROS with Python, C++ and Lisp the libraries rospy, roscpp and roslisp can be used, respectively [10].

The documentation of ROS divides ROS in three different levels of concept: the ROS filesystem level, the ROS computation graph level and the ROS community level [10]. On the filesystem level, ROS is built up of types like packages, repositories and message types. The main unit for organizing software in ROS are packages. They can include nodes, libraries, configuration files or datasets which are dependent on each other and should therefore be organized together. One or more packages can be collected and grouped into repositories. Message types describe the structure of messages that can be published to and subscribed from topics in ROS. There exist primitive message types, e.g. bool, integer or strings[1], but also common message types for navigation, geometry or sensors[2]. These standardized message types have an advantage when it comes to reusing code. Documentation of the different message types is available on the ROS wiki website. If the same message type is used for several subprojects, those subprojects can easily communicate with each other. In addition to that, new message types can be created and implemented if the standard message types are not suitable for a project.

Some of the most important message types for this work are JointState, PoseStamped, Twist and TransformStamped. A message of the type JointState [18] includes a standard header with an ID number, the time in seconds and nanoseconds since the beginning of the UNIX time and the frame ID that this message belongs to. Also the name of the joint, as well as its current position, velocity and effort that is applied in the joint is transmitted. If one of the information is not available, the array can also be left empty. PoseStamped [19] is a message type with a standard header that includes the pose (position and orientation) of an object. Twist [20] is a geometry message that expresses the velocity of an object in linear and angular direction. Both linear and angular

[1] http://wiki.ros.org/std_msgs
[2] http://wiki.ros.org/common_msgs

velocity are presented in vectors of the size three with one entity for each axis. TransformStamped [21] is a message with a standard header which expresses the transformation between a child and a parent coordinate system. It is mostly used by the tf package and is an important input for most localisation and mapping algorithms. A tf message [22] consists of an array of TransformStamped messages. It represents a so-called tf tree, which should include all necessary transformations of a setup. Necessary transformations can be from a map to the base frame of a robot or from a base frame to a specific subcomponent of the robot. The tf message keeps track of all coordinate frames and the transformations between them over time. It lets the user access that information for previous points in time as well as for the current time. Tf can operate in a distributed system and therefore makes the stored information about all coordinate frames of a robot available to all ROS-based components in all computers in a system.

On the computation graph level, different processes in the peer-to-peer network are processing and providing data together. In a peer-to-peer network, nodes have the capability to act as a server and a client at the same time. It is seen as the opposite of a Client and Server architecture, in which nodes can only act as a server or as a client [23]. The concepts within the computation graph level are for example the ROS master, nodes, messages, topics and bags. The ROS master provides different services and information to the nodes. For example, it provides services for naming and registration and keeps track of all publishers and subscribers in a ROS system. The role of the ROS master is to enable the ROS nodes to locate one another. Most often, the ROS master is loaded via the roscore command. Nodes perform computations, as well as publish and subscribe to topics. The data that the nodes publish and subscribe to the topics is structured as messages.

An example of several nodes that publish and subscribe to different topics can be seen in Figure 1. The nodes are indicated with an ellipse shape while the topics are presented in a rectangular shape. The arrows between the nodes and topics represent publishing and subscribing actions. The direction of the arrow defines, if a node was publishing or subscribing to a topic. If the arrow points in the direction of the topic, the message is being published to that topic. On the other hand, if the arrow points in the direction of the node, the node is subscribing to that topic. In this example, there can be seen three nodes and three topics. The robot_state_publisher is publishing a message to tf_static and the rosbridge_websocket is publishing to scan. All three nodes are publishing to the tf topic. In addition to that, the slam_gmapping node is subscribing to all available topics. Therefore, it publishes and subscribes to the tf topic at the same time. This is possible and needed in this case, as the slam_gmapping node is updating the tf message within the tf topic. It needs to be mentioned that nodes can only communicate with each other if the ROS master is running.

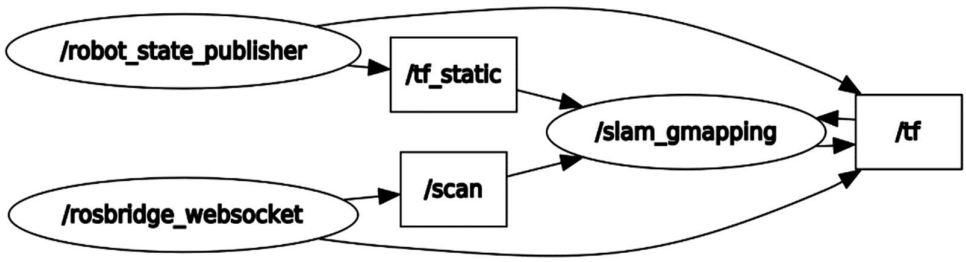


Figure 1. Example of nodes that publish and subscribe to different topics in ROS.

Messages can be saved and played back with so-called ROS bags. The ROS bag format is a logging format which can be used for storing ROS messages in files. The current version of ROS bags, 2.0, contains features that provide the possibility to compress the bag and store the messages by connection to improve the possibilities for republishing the messages from the bag. This way of recording message data can be used for example for evaluating purposes or repeating the same control inputs that have been recorded before. Tools like rosbag and rqt_bag enable the usage and manipulation of the bags [10].

The third level of concept, the ROS community level, enables different communities of ROS users and researchers to exchange their software and their knowledge with each other. The resources in the community level are for example distributions, repositories, the ROS wiki and ROS answers. Distributions are a collection of stacks that can be used for installing ROS. They include sets of software and are available for different platforms and in different memory sizes [24] including different sets of packages. In repositories, own robot software components can be developed and released under open-source licenses by different institutions. Those repositories can then be used, improved and further developed by other institutions and users. The ROS wiki and answers provide the possibility for users to share their knowledge and cooperate to solve problems.

The usage of ROS and its community have been constantly growing in the past years. In the annual metrics report of July 2018 [11], the open robotics team lists 2,204,869 page views for July 2018 with an annual growth of 21%. It is stated that the number of different robot types available for the community with ROS drivers is constantly rising and was at approximately 130 in July 2018.

The growing community and the resulting rise of research and contributions make ROS a desirable tool to use for robotics research and robot development. In this thesis, ROS is combined with the game engine Unity for mobile robots simulation. In the following section, Unity is explained in more detail.

2.2 Unity

Unity is a game engine that was developed by Unity Technologies. It was created in 2005 and has since then become a widely used engine for the development of games. A project in Unity can consist of multiple scenes [6]. Traditionally, those scenes are game levels or menus. They give the opportunity to structure and debug the project in a logical and modular manner. Scenes consist of various game objects, which themselves contain different types of components. The components can define the functionality and behaviour of game objects. For example, components can be joints, colliders or renderers. Game objects can be structured in a hierarchy and thereby be in a parent-child relationship to other game objects. In that way, a single robot can be presented by one game object parent which contains multiple child game objects [25].

Unity provides the user the ability to use an object-oriented framework for scripting in C# and UnityScript. The latter is specifically designed for Unity and modelled after JavaScript. In addition to those two programming languages, other .Net languages can be used with Unity under certain prerequisites [26]. There are two types of object classes in Unity: Runtime and Editor. The Runtime class contains scripts that are used as new types of game components. One or multiple of these scripts can be attached to one or multiple game objects and control their behaviour during game play. Scripts of the Editor classes on the other hand can add menu items to the default Unity menu system.

With such scripts, for example new types of game objects can be added to the Unity editor [25].

Any item that can be used within a project, no matter if it is a scene, a game object or a component, can be turned into an asset [26]. An asset is a representation of that item. It can be either imported from outside of Unity or be made inside of Unity. Files from common CAD software can be imported into Unity and be made an asset. Assets can also be purchased or downloaded for free from Unity's Asset Store. Assets can be reused within one project and also be copied into other projects.

Most actions in the simulation in Unity run at a variable frame rate, while the physics calculations run at a fixed timestep. The ordering and repetition of event functions during one simulation can be seen in Figure 2 [26]. The simulation starts with the initialization of all scripts and events. Afterwards, the physics calculations are executed. These calculations are called fixed update and are based on the fixed timestep. That means that if the fixed time step is less than the actual frame update time, the physics calculations can be repeated more than one time per frame. If the fixed time step is larger than the actual frame update time, it can also happen that the physics calculations are not executed during one frame.

After the physics calculations, the input events from the user are processed. For example, scripts that are triggered by the user with a mouse or keyboard input. When those inputs are processed, the game logic is executed. Afterwards, the scene and GUI (Graphical User Interface) are rendered. As soon as this is finished, the end of the frame is reached. The end of the frame triggers another frame which starts with the physics calculations, if those need to be computed [26].

The physics calculations in Unity are, as common for simulators, done by physics engines that are integrated in the simulation software. There are various physics engines available and many of them evolved from different backgrounds. Because of this, they vary in used approaches and algorithms. In the following sections, some of the physics engines which could be used with Unity and Gazebo are presented and main differences in their approaches are highlighted. Afterwards, several research projects which evaluated and compared different physics engines are sketched out. In the end of this literature review, three common experiments in several of these research projects, which are also interesting for the simulation of mobile robots, are described in more detail.

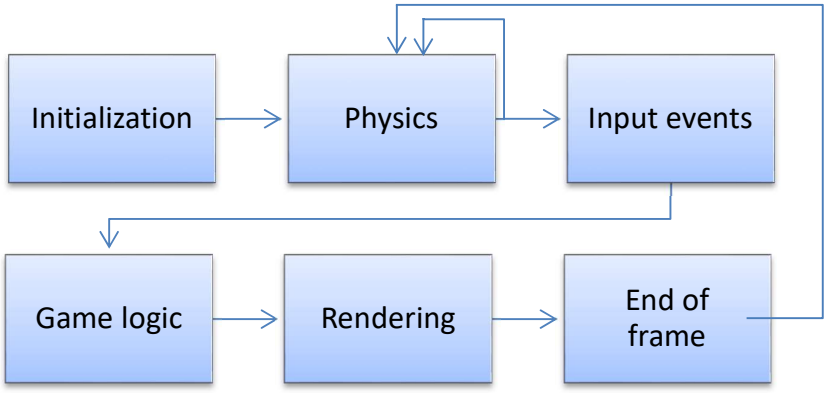


Figure 2. Simulation flowchart. Shows the execution and order of event functions. Created in reference to [20].

2.3 Main characteristics in physics engines

Unity uses the PhysX physics engine by NVIDIA to simulate the physical behaviour of objects and their interaction with the environment. In addition to that, 3rd party physics engines like MuJoCo [27] and Bullet [28] could be implemented if required by the user. This can for example be the case if the 3rd party physics engine has a feature that is not implemented in PhysX or if a certain action requires special physics calculations. The 3rd party engine can then be used instead of PhysX. The simulator Gazebo can switch between four different physics engines: ODE, Bullet, DART and Simbody. This feature evolved over time, as Gazebo started with only ODE as physics engine. Nevertheless, it was stated by Gazebo's developers already in 2004 that the physics engine could be replaced with a better alternative if available [29]. This was made possible due to an internal abstraction layer in Gazebo's layout.

There exist main characteristics in physics engines which determine their performance substantially. One of them is the coordinate representation. Either Cartesian coordinates or joint coordinates can be used. When using Cartesian coordinates, each body has six degrees of freedom (DOF). When representing joints with Cartesian coordinates, constraints are used to limit the type of movement by the joints [30]. That means that for one prismatic joint, which is only able to move in one linear direction, fixed constraints have to be used to limit the six DOF of the joint to one. For systems with no or only few joint constraints, this approach is efficient. If a system has more constraints, more of them can be violated during the process of solving them. This leads to inaccuracies in the computations. In robotics, the joints are often restricted to move in one DOF instead of six DOF, like in the example with the prismatic joint. This leads to many constraints and a high rate of inefficiency when Cartesian coordinates were used. Cartesian coordinates are also described as maximal coordinates [31].

Joint coordinates on the other hand describe joints directly with the amount of DOF that the joints can really move in. So, instead of describing six DOF and then putting constraints on each direction that the joint is not able to move in to construct the joint, only the right amount of DOF is used to define the joint from start. This is more difficult to implement, but computationally more efficient as it is not possible to violate joint constraints [31]. This makes the calculations more accurate for the general robotics case, which usually uses many joints that only have one or two DOF. Joint coordinates are also called generalized [30] coordinates in the community. Joint coordinates are stated to be used by DART ([30], [31], [32]) and Simbody ([31], [33]), while Cartesian coordinates are used by ODE and Bullet ([30], [31], [32], [34], [35]). In [30] it is stated that PhysX and Bullet [35] were in the progress of implementing solutions with Cartesian and joint coordinates at the same time. The article was published in 2015 and the statement regarding Bullet can be confirmed in the Manual of Bullet 2.83 [36].

Another main characteristic of a physics engine is its integrator. The integrator is used for calculating the position of a body in regard to the forces that are acting upon it. It determines the numerical accuracy of a simulation, and common algorithms are implemented with Euler or Runge-Kutta approaches [30]. The integrator of a physics engine can either be explicit, implicit or a mixture of both. When using an explicit integrator, the new state is calculated by using the data of the previous state. When using an implicit integrator instead, the new state is calculated by using the data of the new state. This resolves to be more numerically stable, but also more difficult to implement than the explicit formulation [31].

It has to be mentioned that simulation software in general and physics engines in particular always inherit a trade-off between simulation speed and physical accuracy

[30]. Real world accuracy and instantaneous results are the two extremes on both sides of the trade-off. For example, the results of the numerical calculations become more accurate with more iterations. Those iterations take time and therefore slow down the speed of the simulation. An ideal simulation engine would be able to combine the highest possible speed with the highest physical accuracy.

It is important to understand how different approaches impact the performance of a physics engine. In the case of the coordinate representation, some physics engines like PhysX and Bullet are stated to offer the possibility to define joints in Cartesian coordinates and joint coordinates [30]. The knowledge about the performance benefits of joint coordinates in robotics could influence the user to use the variants with joint coordinate representation when setting up a simulation. The knowledge about the trade-off between simulation speed and simulation accuracy [30] could influence the parameter settings that are chosen for a simulation. It has to be kept in mind that a simulation is always a simplified depiction of reality. The knowledge about the main characteristics of the used physics engine can help to set the boundaries for the validity of the simulation results.

2.4 Research in physics engines

Another approach to rate the performance of physics engines is to compare them to each other. This has been done in several research projects so far. Due to the complexity of the engines, their tuneable parameters and the rapid ongoing development, those projects often struggle with a reasonable choice for a general "best" physics engine. Nevertheless, most engines come from a different background and therefore have their own, specific direction and focus. Those backgrounds and focuses of the physics engines and their underlying algorithms lead to strengths for specific tasks ([30], [34], [37], [38]).

In a survey based research by Ivaldi and Peters [35], a classification of the simulation tools was presented. Simulation tools were divided into physics engines and system simulators, like Gazebo, Unity or V-Rep. The authors further classified physics engines regarding to their approach in coordinate representation. System simulators on the other hand are more complex than physics engines. They include and are based on a physics engine, but can also model and simulate sensors and robotic interfaces, as well as provide advanced interfaces. As a conclusion, it was stated that none of the physics engines was overall superior to others. For the system simulators, Gazebo, which worked in a ROS environment, dominated the other open source simulators in the area of humanoid robotics. V-Rep was the preferred commercial simulator, even though it has to be mentioned that, according to the results, V-Rep and Webots were the only not open-source simulators in the survey. The authors chose to evaluate simulation tools with a survey, as the different requirements and features of all the engines made them hardly comparable on a common base. As a main request from the robotics researchers, they stated that better physics engines, therefore more realistic simulations, the same code for both real and simulated robots and more open-source software were desired. The four important features for a simulator for robotics researchers were stated to be the stability of simulation, speed, precision and accuracy of contact resolution. The two most important criteria for choosing a simulator were a high compliance to reality and open source availability of the code.

On the contrary of this survey-based feedback, several experiment-based comparisons of physics engines could be found in different research projects. The aim of these

projects was often to find the strengths and weaknesses of the physics engines in order to be able to determine the best possible engine for a current use-case.

A comparison of physics engines with a focus on contact simulation was presented by Chung and Pollard [32]. Here, the parameters of each engine were tuned in order to match an analytical solution in a first step. This was done to create a common ground and base for the comparison of different physics engines. As the different physics engines didn't have the same parameters and settings to tune, it was difficult to ensure that they had the same basic conditions. Tuning the parameters to match an easy, analytical solution was therefore a process to try to set the different physics engines on the same basic conditions. After the tuning process, a more complex simulation was done and the results of the physics engines were compared against each other. The chosen experiment was a rolling cube in two different scenarios. The simpler version included the cube rolling downhill in one direction while the more complex version simulated the cube rolling on flat ground in various directions by applying varied magnitudes of forces on the cube in different directions. The authors stated that the parameter settings clearly influenced the results of the experiments. In addition to that, the physics engine that was performing especially predictably and smoothly in the simple simulation, namely MuJoCo, did perform worse in the more complex experiment. Other physics engines, that performed worse in the first experiment, produced more predictable simulations during the more complex simulation.

Erez, Tassa and Todorov [30] compared several physics engines with regard to speed and accuracy. For measuring the accuracy, a consistency measurement was chosen. As reference for the consistency, the trajectory of a simulation with a timestep of $^1/_{64}$ ms was chosen. The accuracy was then calculated by accumulating the deviation of the trajectory in small integration intervals. Four different test scenarios were built and simulated with all tested physics engines. The test scenarios were an arm grasping an object, a falling humanoid, a planar kinematic chain and a capsule collision test. In addition to the speed-accuracy measures, measurements for the conservation of kinetic energy, angular momentum and linear momentum were done in regard to the speed of the physics engine. It was concluded that each physics engine performed best for the area it was designed for with no clear leader in all tasks and areas.

In a research by Roennau and Sutter [38], Bullet, ODE and PhysX were evaluated with a focus on the applicability for the dynamics of walking robots. For this purpose, eight different experiments were designed and evaluated for all physics engines. It was concluded that no physics engine outperforms all other engines in all aspects. While PhysX was superior in collision detection and constraint stability, Bullet could offer a much more precise friction model.

It was stated that physics engines often use an approximated friction pyramid instead of the more precise friction cone model. To investigate the used friction model, two boxes were placed onto each other with defined friction coefficients between them. Gravity was then applied in the direction of the negative z-axis. Step by step, the direction of gravity was rotated around the x-axis of the world coordinate system. The angle of gravity where the upper box started to move, was notated. This procedure was repeated for 360° around the z-axis of the world coordinate system in specified steps. If a friction cone was used, the angle around the x-axis should be the same for every position around the z-axis, resulting in a circle when looking in z-axis direction. Otherwise, the angle on the x-axis would deviate from each other and result in something different from a circle. While Bullet had a friction cone in this experiment, PhysX showed a friction pyramid with comparably huge deviations. Due to the high accuracy

of the friction model, Bullet was used for the follow up project of this research. This was chosen as the friction model was seen as essential for walking robots, when focusing on the slipping of the legs. An independent implementation made it possible to exchange Bullet with another physics engine rather easily if desired [38].

Boeing and Bräunl [34] evaluated and compared several physics engines in a research published in 2007. Here, the overall performance of physics engines was evaluated in regard to six essential factors that evolve out of the construction and algorithms of a physics engine and determine its performance. Those six essential factors were the simulator paradigm, the integrator, the object representation, the collision detection and contact determination, the material properties and the constraint implementation. A physics engine abstraction system, called PAL, was used for implementing and simulating the test scenarios with the different physics engines. It was stated that some of the engines used for research and simulation in the community were not appropriate for simulating common problems.

For example, in the test for the integrator performance, a body was accelerated through gravity and its position was compared to theory. In this test the Newton Game Dynamics physics engine had an error of approximately 45 times of the other physics engines. This was due to forced velocity damping in the Newton Euler integrator, but had definitely to be taken into account and made the Newton Game Dynamics physics engine an inappropriate choice for certain tasks. In this research, it was again concluded that none of the physics engines performed best for all the tasks and each of them had their own strengths.

2.5 Experiments for physics engine evaluations

There are some experiments which are used for evaluating physics engines in multiple research projects. These are explained in this chapter, as variants of them are used in Unity to evaluate its basic performance and judge the influence of several parameters on the simulation outcome.

One commonly used test scenario, especially in the area of gaming engines, is regarding collision detection. There exist variations for the conduction and evaluation of this scenario in different research projects. All variants of this scenario are not feasible for being tested in reality. In one of the variations, capsules are placed in the air in various positions and orientations [30]. As soon as the simulation starts, gravity is accelerating the objects towards the ground. This experiment is repeated with various simulation speeds. The slowest simulation with the highest number of physics calculations is taken as a reference for the faster simulations. The error in trajectory compared to the trajectory in the most accurate simulation is compared with other physics engines. The setup of this variation of the collision detection experiment can be seen in the left picture in Figure 3.

Another variant is using multiple spheres which fall into an inverted square pyramid made out of a mesh [34]. During the simulation, the penetration of the spheres with the pyramid is measured. The sum of all penetrations is calculated and plotted against time. This method is used for physics calculation frequencies of 100 Hz and 15 Hz in various physics engines and the results are compared with each other.

A third version of the collision scenario is testing the collision of boxes, capsules, cylinders, spheres and triangular meshes with a triangular mesh height field [38]. A mesh is an object of which the surface is made out of a finite element method. The triangular mesh height field here is used as an uneven floor. Small, medium and large objects are

tested in different measurement runs. Multiple instances for each of the collision shapes are created and tested in three physics engines. The results are evaluated by pass or fail. The experiment is rated as failed if objects get stuck in the triangular mesh floor or fall through it.

A different test scenario is used for measuring the integrator performance of a physics engine. The integrator of a physics engine calculates the position of a body in regard to the forces acting on it. The different approaches for integrator algorithms and their effects were discussed in Section 2.3. In the simulation, an object, for example a sphere, is created and a force is applied. This can be for example gravity [34] or gravity in addition to a horizontal force (similar to throwing a ball) [31]. In contrast to the collision detection test, this scenario can be compared to reality or theory. The relative error to reality or theory can be calculated for every physics engine and the results compared with each other. An example of a simple setup with just gravity acting on a sphere can be seen in the intermediate picture in Figure 3.

A third very common experiment is the stacking test. In this experiment, multiple boxes are stacked onto each other. The number of stacked boxes is increased until the stack collapses. The maximum number of stacked boxes without a collapse is compared for different time steps or different physics engines. The difficulty of this task for the physics engine is increased with the number of boxes, as more friction forces and normal forces for preventing penetration have to be calculated [38]. The inaccuracy of these calculations leads to drifting in the position of the bodies and therefore in a collapse of the stack. An example of the stacking test can be seen in the right picture in Figure 3. Variants of this test are performed in [31], [34], [38] and [37].

In the previous sections, the usage of Unity as a simulator was described, the setup of ROS and Unity was explained, main characteristics in physics engines as well as research for evaluating them was presented and common experiments in those evaluations were introduced. This background information is important for understanding the setup of the simulation of mobile robots with Unity and ROS and rating the simulation results in regard to accuracy.

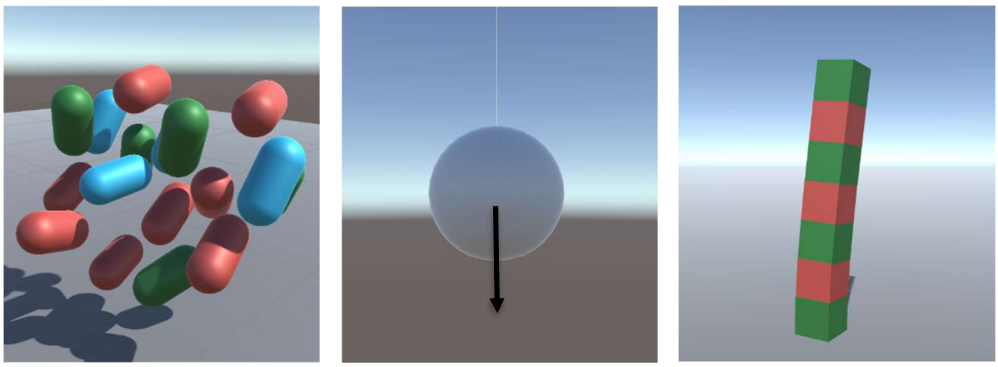


Figure 3. Common experiments for evaluating physics engines. On the left side is the collision test with capsules. In the intermediate picture, a integrator test with gravity acting on a sphere can be seen. In the right picture, a stacking test is performed.

3 Method

The method for the development of this thesis was as follows. Firstly, a literature study was conducted. It consisted of four main parts: the usage of Unity as a robotics simulator in research, ROS, Unity and the physics engines used in Unity and Gazebo. The first part should reflect if Unity, despite its background as a game-engine, has been used as a robotics simulator so far. The second and third part of the literature study should give insight into the functionality and structure of ROS and Unity. The last part was done to create an overview of the physics engines and to describe how they were structured and have been evaluated so far.

The insights from the evaluation of the physics engines in the literature review were used when determining the test scenarios. Basic test scenarios similar to the ones used for evaluating physics engines were used. Additionally to those basic scenarios, more advanced qualitative and quantitative scenarios which included the usage and analyse of a TurtleBot2 model were determined with respect to important functionalities of mobile robots. The Unity environment was set up in such a way that the same control as in the Gazebo simulations and in reality could be used. For doing that, ROS and RosSharp was used to connect Unity with the control script that can be used for the reality, Unity and Gazebo. The used open-source model of the TurtleBot2 with a Kobuki base was modified to correspond better to the robot used in reality. This model was tested and partly corrected to conform to reality.

Test scenarios were created in Unity with the previously modified and tested model of the TurtleBot2. While creating and testing those scenarios, the parameter settings of the physics engine were investigated and the most interesting and important ones for the setup of mobile robotics were identified. After the test scenarios have been created, measurements with the simulation as well as with the real robot have been conducted. For the quantitative benchmarks, those measurements have been repeated for a certain amount of times to investigate the distribution.

In addition to those benchmarks, a SLAM case study was conducted. It aimed to investigate the possibility to employ SLAM algorithms within Unity simulations of mobile robots. In order to successfully employ SLAM algorithms, additional scripts for the timing between ROS and Unity, as well as the creation of the tf tree had to be developed. The SLAM simulation was tested in a labyrinth in Unity.

As a next step, the data from the measurements was evaluated. For that purpose, Python scripts were written. In those Python scripts, plots to present the chosen evaluation parameters were developed. The results of the measurements with the real robot, with Unity and Gazebo were discussed and compared to each other. Unexpected results were examined in more detail to search for an explanation. In the end, all the results and the experiences during this thesis work were summarized and brought to a conclusion of Unity's performance when simulating mobile robots with ROS.

4 Simulation setup and scenarios

For the conduction of the experiments, several parts had to be prepared and investigated in advance. This chapter describes the preparations that were done and focuses on important setups of the experiments. In the first section, important settings in Unity which are essential for the performance of the simulator are described. The subsequent section focuses on the open-source project RosSharp, which is used to connect Unity with the ROS environment. The third section describes the model of the TurtleBot2, which is used for the simulation, and what modifications were made. The last section describes the chosen test scenarios, why they were chosen and how they are evaluated in detail.

4.1 Unity settings

Due to the complexity in detailed simulations of reality, Unity comes with a variety of variables and settings that influence the behaviour of the simulation. These parameters have to be set properly to enable the robot to perform in a realistic way. During the work process of this thesis, some of these parameters were identified to have a great impact on the simulation results. Small changes can result in huge performance differences. The most important parameters were discovered through testing and parameter tuning during the setup of the experiments. Those are mentioned and described here. A summary of the parameters can be found in Table 1. Unity does not set fixed units for most of the parameters, except for the different time steps. Units are defined to be units and can be set as mm, cm, inch or any other preferred unit of measurement. Nevertheless, Unity Technologies recommends to assume one Unity unit to be 1m, as this would fit best for most physics systems [39]. References and official descriptions for these parameters can be found in the Unity manual [26]. A complete table of the chosen default parameters for all test scenarios can be found in Appendix 1.A.

Table 1. Selected parameters with their default values in Unity.

Parameter	Default value
Fixed timestep	0.02 s
Maximum allowed timestep	0.1 s
Default solver iterations	6
Default solver velocity iterations	1
Default contact offset	0.01
Static friction	0.6
Dynamic friction	0.6
Friction combine	Average
Enable adaptive force	False

In the time settings of Unity, the fixed timestep can be chosen. This variable determines the interval at which the physics calculations are performed. The default value for this variable is 0.02 s, so that the physics calculations should be performed at 50 Hz. A smaller fixed timestep generally gives more accurate results, as more calculations lead to outputs with a smaller numerical error. If on the other hand the fixed timestep gets

too small, the simulation can become numerically unstable more easily, as errors accumulate faster. How the fixed timestep correlates to the frame rate for the Unity simulation and how the execution of the physics calculations is ordered, is discussed in Section 2.2.

The maximum allowed timestep is closely related to the fixed timestep. If many objects are simulated and the physics calculations take a lot of time, it can happen that the physics calculations overload the CPU capacity and result in a drop of the frame rate. As Unity is a game engine and tuned for a "smooth" user experience, significant drops of the frame rate and resulting noticeable effects and ripples in the game are undesirable. If the physics calculations take longer than the maximum allowed timestep, the physics engine would stop until the frame update is finished. This would reduce the accuracy of the physics calculations and therefore the accuracy of the whole simulation. Because of the accuracy drop, this is an important variable for the experiments of this thesis. The default value by Unity for the maximum allowed timestep is 0.1 s.

In Unity's PhysicsManager, parameters for the physics engine can be set. For example, the default solver iterations define how often the solver iterates in each fixed timestep. If this value is increased, the solver outputs are more accurate, but the calculations take more time. Unity states in its manual that this value should be adjusted if the default timestep is not used or the configuration is especially demanding. This value is set to 6 by default.

A similar parameter is the property for the default solver velocity iterations. It defines the iterations in each fixed timestep for the calculation of velocities. By default, this value is set to 1. Changing the default solver velocity iterations and the default solver iterations does not affect the physics calculations for already created objects with a "Rigidbody". To change the amount of iterations for already created objects, their internal parameter "solverIterations" or "solverVelocityIterations" has to be modified, respectively. In respect to the model used here this means that a change in the default solver (velocity) iterations does not take effect unless it is made before importing the model into Unity. For changing the values after importing the model into Unity, the values have to be changed via a script, as the internal solver parameters are not accessible via the GUI. Examples for those scripting methods can be found at the scripting API of the "Rigidbody" in Unity's manual [26].

A parameter that has to be handled in a similar way is the default contact offset. This variable describes the distance that is used by the collision detection system to generate collision contacts. If the distance between two colliders is less than the sum of their contact offsets, a collision contact is generated. That means that two objects do not necessarily have to have a physical contact in order for a contact to be detected by the physics engine. If two objects do not physically collide, but the distance between their colliders is less than the sum of their contact offsets, this is detected as a contact anyway. If the contact offset is set too low, the objects can more easily start to jitter [26]. Again, the default contact offset can only define the contact offset of game objects that are generated after it has been set. If the contact offset of an already existing game object should be changed, it has to be done with a script. The default value by Unity is 0.01.

An important setting for simulations, especially if it comes to mobile robot simulation, is the friction between different objects. A game object can only have a friction value if it has a collider component. A collider component is an invisible shape that defines the collision surface of a game object. If a collider component is available in a game object, the friction values can be set via a physics material that can be referenced

in the collider. In the physics material, the dynamic friction, static friction and bounciness can be defined. In addition to that, the friction combine and bounciness combine have to be set. The default values will be set to all colliders which do not inherit a customized physics material. They are set to 0.6 for dynamic and static friction, 0.0 for bounciness and average the combination methods by default. The friction combine defines how the friction values of two colliding objects are combined. Apart from average, it can also be set to minimum, maximum and multiply. If the friction values are too low for a wheeled mobile robot, the wheels are more likely to slip and the trajectory of the robot will not be according to the control inputs that are given to the robot. Unity states in its manual that the friction model of PhysX is tuned for gaming applications and does therefore not necessarily need to be very accurate in regard to real-world physics.

Another parameter is used for enabling adaptive forces. Unity states in its manual that an activated adaptive force results in more realistic behaviour for stacks and piles of objects. The usage of adaptive forces is disabled by default. In a blog post [40], Unity states that the parameter adaptive force is used to compensate numerical errors when stacks are simulated and that it decreases the stability of the game to a high extent. Because of this trade-off between numerical accuracy for high stacks and the game stability, the usage of adaptive forces seems to be advantageous only when it comes to significant high stacks of objects which need to be modelled very detailed.

4.2 RosSharp

For evaluating Unity and comparing it with Gazebo, the same control for both simulations and the real robot should be established. Gazebo has a ROS interface and can therefore easily be connected to topics with navigation and sensor messages and be controlled by them. ROS can also be installed on the TurtleBot2 and enable the robot to be controlled via a remote connection. Due to these circumstances, ROS is seen as a reasonable environment for controlling both simulation models and the real robot for these experiments. As at the time of this thesis the developer platform of Unity was only available on Windows and Mac while ROS was only available on Linux, a bridge for connecting Unity with ROS was needed. An open-source project of Siemens on GitHub, called RosSharp, is providing the possibility to connect Unity with ROS. It mainly consists of an URDF-transfer and a rosbridge client [41].

The URDF-transfer enables using the same URDF file, which describes a model in ROS and Gazebo, in Unity. In an URDF file, the different links and joints of an object as well as their parameters can be defined. Among the parameters are the dimensions of the parts, their weight, inertia, collision geometry, appearance and other physical properties. The collision geometry represents the geometry which is used by the physics engine to determine if the part is colliding with another object. The closer this geometry is to the real geometry, the more accurate the collision detection can work. On the other hand, the simpler the collision geometry, the more easy and efficient the computation for the physics engine will be. The other physical properties include the friction, contact stiffness and damping of objects. Those parameters for friction, contact stiffness and damping are not transferred to Unity directly with the URDF-transfer of RosSharp. They have to be edited in the Unity model after transferring the URDF from ROS.

The rosbridge client is a collection of several scripts which enables the connection of ROS and Unity with a rosbridge [14] via the network. A rosbridge is a protocol that provides a JSON interface to ROS. It enables clients to send JSON to connect to ROS software by publishing and subscribing to ROS topics, calling services and more. The

rosbridge client in RosSharp includes scripts for subscribing and publishing basic ROS messages like joy, odometry and many more. Joy messages include the state of a joystick's buttons and axes, while odometry messages include the estimate of a robot's position and velocity relative to a coordinate system. More information about ROS messages can be found in [42]. The control message which is used for controlling the movement of the real robot in this thesis is a twist message. This message defines the linear and angular velocity that the robot should achieve. Subscribing to those twist messages and calculating the resulting angular velocity for the wheels of the TurtleBot2 was not included in the RosSharp scripts. A script for this feature had to be written and implemented in Unity and the RosSharp environment in order to use the same scripts for the real robot and the simulated Unity robot. The resulting angular velocity is set as target velocity for the motor in the actuated wheels. This script can be seen in Appendix 1.C.

The setup of the working environment with Unity and ROS during a simulation can be seen in Figure 4. A computer with a Windows operating system is used for the simulation experiments in Unity. On top of the windows host OS, a virtual machine (VM) with an Ubuntu OS is running. The two OS are connected to the internet via a proxy server. Information about the software version and hardware specifications for all experiments can be found in the Appendix 1.A.

ROS is installed on the Ubuntu VM. On this VM, the ROS master is running. All nodes of the Ubuntu side, as well as the scripts in Unity through the rosbridge in RosSharp connect to this master. In addition to that, the URDF file for the robot model is available and can be read and used for the URDF import with RosSharp to Unity. When a simulation is running, the control script is executed on the VM. The control scripts consist of nodes which publish to control topics. The messages in the control topics are subscribed by Unity through the rosbridge and corresponding scripts in RosSharp. These messages are then used in Unity to control the robot during the simulation. Simultaneously, scripts in Unity are reading simulation outcomes like the angular velocity of the wheels of the robot. These scripts then transfer the data into ROS messages. While doing this, certain messages like velocity-arrays have to be adjusted in terms the used coordinate systems in the different software. Unity uses the left-handed coordinate system [26] while ROS is using the commonly used right-handed coordinate system [43]. Functions for converting vectors and quaternions from ROS to Unity and

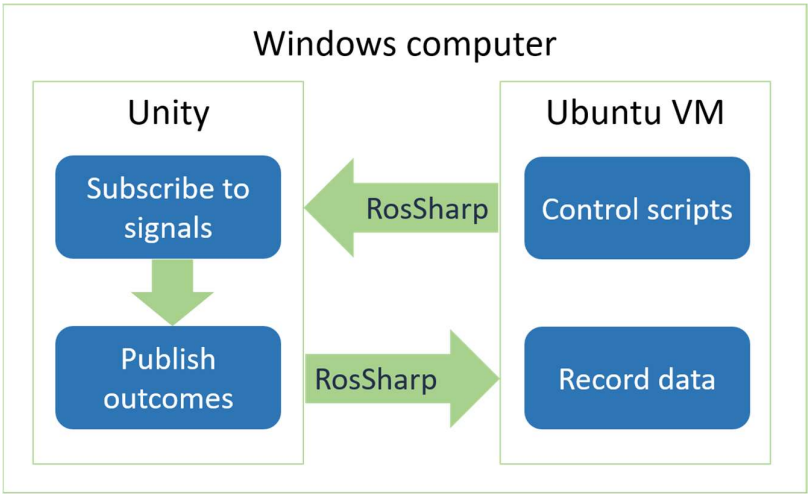


Figure 4. Setup of Unity and ROS with RosSharp for the simulation.

vice versa are included in RosSharp [44]. These functions can be used when subscribing or publishing to new messages. They then have to be implemented into the scripts in Unity before processing the subscribed messages or when preparing the data that is acquired in Unity for publishing.

After the conversion is finished and the information is stored in the right message types, they are published to topics via the rosbridge and RosSharp. On ROS-side, rosbags are recorded which subscribe to all necessary topics. These rosbags can later be used for the evaluation of the simulation.

4.3 Robot model

For this work, a modified version of the TurtleBot2 with a Kobuki base was used. This was accordingly to the TurtleBot2 that was available at the laboratory. The Kobuki base consists of the control logic, internal sensors and mechanical units of the robot. It has two actuated normal wheels on the left and right side of the TurtleBot2 and two unactuated normal supporting wheels in the front and rear. On top of the base, three plates are stacked and connected with poles. The distance between the plates is increasing towards the top. On top of the second plate and below the third plate, a camera is installed on extra poles. A model of the original TurtleBot2 in Unity is shown in the left picture in Figure 5. This model was generated with the open-source TurtleBot2 [45] and Kobuki model [46], as well as the URDF import plugin of RosSharp.

The open-source URDF file of the TurtleBot2 was edited to match the modified robot for the real test scenarios. For that, unused parts were removed. The unused parts in this case were the original Kinect camera, the second and third plate, all the corresponding poles and the links for the base-internal gyro and cliff sensors.

Cuboids in the size of the used laser sensor (Hokuyo UTM-30) and camera (RealSense D435) were added. The cuboids were matched to the real sensors in regard to size and weight. The inertia of the models was calculated according to cuboids with the dimensions and weight of the sensors. This solution was used as open-source models of the sensors were not available and only the position, size, weight and resulting momentum was of importance for the dynamic behaviour of the simulated robot. As models of additional cables and fixtures were not included in the original URDF file of the TurtleBot2, the mass of these was determined by weighting the real robot and calculating the difference to the plain model with all parts but without cables or fixtures. This

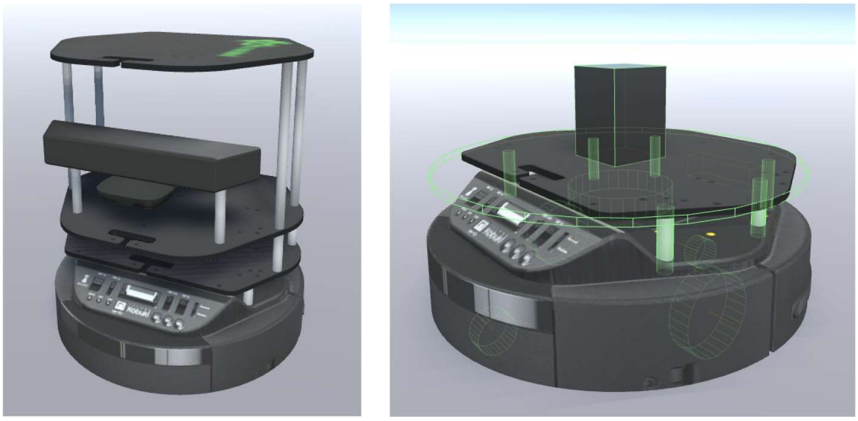


Figure 5. Model of the original (left) and modified (right) TurtleBot2 in Unity.

mass was added with an additional weight in form of a cylinder on top of the base of the robot. The modified model in Unity can be seen in the right picture in Figure 5.

When working with the open-source URDF model of the TurtleBot2, several mismatches in the URDF file were discovered. These mismatches were corrected before using the model for the test scenarios. For example, the z-position of the supporting wheels at the base of the Kobuki model was set to 0.007 m for the front wheel and for 0.009 m for the rear wheel. This mismatch caused the robot to wiggle up and down slightly when starting to accelerate or to decelerate.

Another correction in regard to the supporting wheels of the TurtleBot2 was made. In the URDF model, the joint of these wheels were set to fixed joints and their friction coefficients were set to zero. Because of that, the wheels of the corresponding model in Unity could not rotate but were fixed in position and orientation in regard to the base link. The friction coefficients were not inherited when importing the URDF model to Unity. This combination of fixed joints and the default friction coefficient in Unity led to increased friction, as those wheels were just sliding and not rotating on the ground. The supporting wheels of the real robot were normal wheels which could rotate around their z-axis when the robot was moving forward or backward. To match the real robot more closely, the joints of the supporting wheels in the URDF model were changed from fixed joints to continuous joints.

A setting in the URDF that turned out to be of major importance for the behaviour of the URDF-model based robot in Unity was the inertia of the parts. Initially, the inertia of the additional sensors in the used URDF model was set to 1 kgm^2 for all axis. This was not discovered until some of the TurtleBot2 robots in the simulation turned to be numerically unstable. When starting the simulation they began to vibrate. Those vibrations accumulated until the robot started to move, jump and burst in an unforeseeable way. This behaviour improved significantly due to removing the additional sensors from the robot model in Unity.

When taking a closer look to the URDF model, comparably high values for the inertia were discovered in several parts. The inertia around the x-axis for the camera for example was $5{,}235 \cdot 10^{-5}\ kgm^2$ instead of 1 kgm^2 when computing the inertia according to the formulas for a cube with the same dimensions and mass of the real camera. When adjusting the values to the computed inertia values, the numerically unstable behaviour vanished. The extremely high inertia values affected the dynamical behaviour of the robot. As soon as it started to vibrate – for example because of a collisions with another object – the inertia values caused the vibrations to accumulate and increase themselves.

To have a correct robot model is of essential importance for the simulation results. The numerically unstable behaviour of the robot and its "wiggling" movement illustrate that importance. Only a validated robot model where the limitations are known can be successfully used for realistic simulations of robotic behaviour. The setup of the used URDF model as well as all dynamic parameters are listed in Appendix 1.B.

4.4 Test scenarios

For the evaluation of Unity in regard to reality and a comparison to Gazebo, test scenarios need to be defined and implemented. Different levels of complexity are covered by using basic tests without the robot and ROS for the general performance of Unity in certain aspects and more complex scenarios with a TurtleBot2 model and a connection to ROS. The more complex scenarios can be further divided by distinguishing

between test scenarios with a qualitative and a quantitative evaluation. The test scenarios are explained in detail in the following subsections.

4.4.1 Stacking

As a basic experiment, a test with stacking boxes onto each other is performed. Variations of this experiment have been performed in several research projects for evaluating physics engines, see Section 2.5. In this experiment, boxes are stacked onto each other.

Due to inaccuracy in the calculation of friction forces and normal forces, the boxes move slightly during the simulation. This movement is – apart from the algorithms that are used in the physics engine and the characteristics of the boxes – depending on the frequency of the calculations and the amount of stacked boxes. If the used physics engine, the characteristics of the boxes and the frequency of the calculations do not alter, an increasing amount of stacked boxes results in a collapsing stack at some point. This amount of stacked boxes for each calculation frequency can be compared for different physics engines.

The experiment is simulated in Unity and Gazebo. In both simulators, boxes with the dimensions of 1m x 1m x 1m and the mass of 1kg are used. The time step that defines the frequency of the physics engine's calculations is varied. If the stack does not collapse within 20 seconds, the test is rated as passed. Unstable stacks can collapse after a very short time or escalate for some time before collapsing. In choosing the time threshold to be 20s, some stacks are rated as stable even though they would collapse in the end. This was chosen to simplify and speed up the evaluation of the benchmark, similar to [38]. As an evaluation parameter, the minimum amount of boxes that cause a collapse against the used time step is compared for Unity and Gazebo. The stacking test in the Unity simulation environment can be seen in the left picture of Figure 6.

4.4.2 Collisions

Another basic experiment is evaluating the simulator's ability to detect collisions with multiple objects. This scenario is often used to evaluate the performance of game engines and different variants of it have already been described in Section 2.5. For the purpose of this thesis, a mixture of those scenarios is chosen to shed light on the important and affecting parameters regarding collisions in mobile robot simulation.

For the first variant, which was presented in [30], 27 randomly oriented capsules are simulated falling on the floor. While the capsules are randomly oriented for the evaluation of each physics engine, their pose for the beginning of each simulation within that physics engine stays the same to ensure the comparability. The results show graphs with

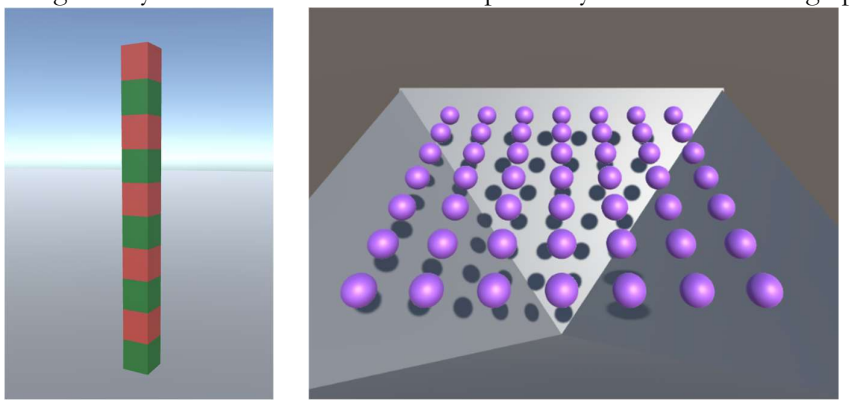


Figure 6. Stacking test (left) and collision test (right) simulated in Unity.

the speed of the simulation plotted against their accuracy. The accuracy is measured by comparing small steps in the trajectory against the trajectory of the most accurate simulation. The trajectory itself does not give much insight into the collision detection, as it depends on the contact point when colliding. Therefore, this variant was not chosen as a test for the work of this thesis. Instead, a greater weight is set on the two other variants. There exist basically two interesting cases when it comes to failures in collision detection for mobile robotics simulation.

The first scenario is when no collision at all is detected by the simulator or the physics engine, respectively. If this is the case, the objects move through each other without interacting. This can happen out of two main reasons. Firstly, if the according layer collision parameters in the PhysicsManager are set by the user so that no collision should occur between those objects. [26] Secondly, if the speed of the object in combination with the fixed timestep leads to a missing recognition. If the objects are too fast or the collision is checked too rarely in the physics engine, it can happen that objects collide, move through each other and exit the collision before the collision status is checked again.

The second scenario is if the collision was detected, but the objects have penetrated each other. This behaviour is dependent on the speed of the objects, their offset collision value and the fixed timestep. To test this scenario and get an impression of the penetrations, spheres are positioned above a top-down, four-sided pyramid, similar to [34]. The simulation of the scenario in Unity can be seen in the right picture of Figure 6.

With the start of the simulation, gravity accelerates the spheres towards the pyramid from which they recoil and then collide into each other. During the simulation, the penetration of spheres into the pyramid and each other is constantly computed. For computing the penetration between a sphere and the pyramid, the distance between the sphere and all four planes of the pyramid is calculated. For computing the penetration between two spheres, the distance between the origins of two spheres is calculated and subtracted from the sum of their radii. The maximal penetration for each step in time is stored and all maximal penetrations are saved in a file at the end of the simulation. The simulation lasts for three seconds in total, as all spheres reach approximately their final destination at that time. The fixed timestep is varied and the statistical results of each run are put into a box diagram.

4.4.3 Integrator

The integrator scenario is a basic benchmark used to evaluate the integrator performance of a physics engine. This means the performance of the physics engine when it comes to calculating a body's position in regard to the forces acting upon it. It is already described as being used in physics engine evaluations in Section 2.5. For this evaluation, a sphere with a radius of 0.5m and a mass of 1kg is positioned at the origin of the world coordinate system. Gravity of 9.81 $^{m}/_{s^2}$ and no air resistance is applied.

In theory, the position of the sphere results out of the applied force, e.g. gravity, and the time of the free fall. It can be computed with:

$$z = \frac{1}{2} \cdot g \cdot t^2$$

The simulation is run for 10 s and then paused. The error in absolute position of the sphere is calculated in regard to theory. The fixed timestep is altered and the results are plotted in a scatter diagram.

4.4.4 μ-split

A more complex qualitative test scenario that was implemented is focusing on the friction between the wheels and the floor. Two materials with different friction coefficients are placed on the floor. The TurtleBot2 model is placed between those materials in such a way that one actuated wheel experiences a higher friction force from the ground than the other one. Both differential drive wheels of the TurtleBot2 model are actuated with the same angular force.

In theory, this difference in the friction force should have an effect if the friction of one wheel was so small that the tyre starts slipping and has no grip on the ground. If that was the case, the angular force on that tyre does not result in a forward force for the robot anymore. If the angular force on the opposite tyre still results in a forward force for the robot, the robot starts rotating around its z-axis towards the slipping tyre. If the circumstances are in a way that the tyre is slipping, it behaves as if that tyre is actuated with less or no force than the other tyre.

In the simulation, this rotation should be seen with sufficient low friction coefficients for one wheel. For the qualitative evaluation of this test scenario, the simulation is executed in both simulators and the trajectory of the robot with the same control inputs for different friction coefficients is measured and compared. The setup of the μ-split scenario can be seen in Figure 7 on the left side.

4.4.5 Ramp

For another qualitative test scenario, the TurtleBot2 is placed on a ramp with different slopes. The robot is controlled to drive up the ramp with a speed of $0.2\ {}^{m}/_{s}$. The robot is placed on the ramp from the start of the simulation. To prevent the robot from rolling down the ramp backwards before its wheels are actuated, a wall is placed behind the robot.

The robot is not placed in front of the ramp as it could not drive over the beginning edge of the ramp. This is because of the modelling of the TurtleBot2 in the URDF [45]. The spring suspension on the actuated tyres is not modelled in the URDF. As soon as the robot begins to climb the ramp, the actuated tyre does no longer have contact with the ground. The supporting wheel in front already has contact with the ramp while the supporting wheel in the rear still is in contact with the floor. The actuated wheels, which are positioned somewhere between the supporting wheels, hang in the air, respectively only touch the ground slightly without a sufficient amount of normal force acting upon them. In the case of the real robot, the actuated tyres are pressed on the ground as they

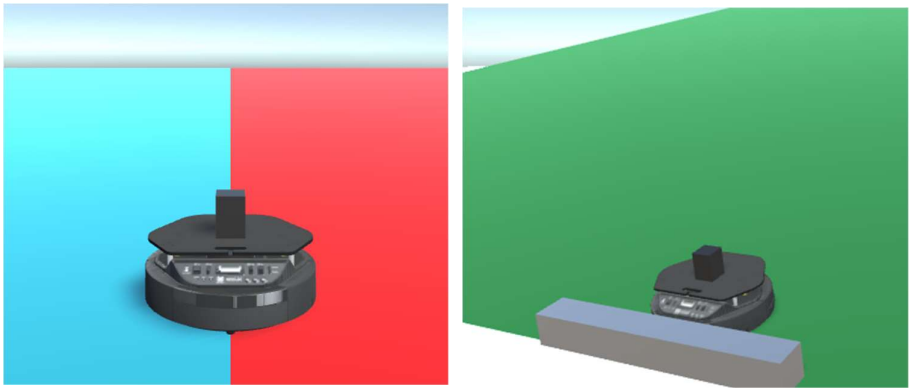


Figure 7. μ-split scenario (left) and ramp scenario (right) in Unity.

are mounted on a spring suspension. Because of this lack in detail of the simulation model, no forward force can act on the robot when starting to climb the ramp and the robot is stuck on the edge.

To judge the simulators apart from this error in the model, the robot is placed on the ramp from the start of the simulation. The setup can be seen in the right side of Figure 7. For the first simulation, the angle of the ramp is set to 5°. After each simulation, the angle is increased by 5° and the measurement was repeated. At a slope of 30°, the angle increment is set to 2° and the measurements are repeated until 36°. These slopes are chosen as they showed the difference in the robot's behaviour in Unity clearly. For evaluation purposes, the trajectory of the TurtleBot2 with different slopes is measured and compared. The TurtleBot2 is expected to not be able to drive up the ramp anymore at some point when increasing the slope.

4.4.6 Friction turn

Nowadays, vacuum cleaning robots are a common sight in private households. Those are designed similar to the TurtleBot2. The friction turn is supposed to be a qualitative test which is related to the working environment of those vacuum cleaning robots. While cleaning, those robots often slide along objects or walls. This sliding can cause the mobile robot to turn around its z-axis, depending on the friction forces which are exerted by the contact between the robot and the objects. For this test, the TurtleBot2 drives against an object in an angle of 20° between the surface of the object and the moving direction of the robot. The test is passed if the robot is rotating towards the object when colliding with it. The behaviour of the robot during the measurement is observed, recorded and the rotation is judged visually.

4.4.7 Basic drive

A test scenario which can be evaluated in a more quantitative way is the basic drive test. Here, the TurtleBot2 model is used and the robot is controlled by a Python script on ROS. The connection between Unity and ROS is established via RosSharp, as described in 4.1. Twist commands, which consist of an angular and linear target velocity, are used.

The TurtleBot2 starts at the origin of the world coordinate system. It then starts to drive with a target speed of 0.2 $^{m}/_{s}$ for 20s. After 20s, the TurtleBot2 is stopped and waits for 5s. Afterwards, the robot is driving backwards with the same velocity for 20s. It is then again stopped and a waiting time of 5s is applied. This procedure is repeated with a speed of 0.25 $^{m}/_{s}$ and 0.3 $^{m}/_{s}$ afterwards.

The absolute position and orientation of the robot are measured in regard to the world coordinate system. In reality, this is done by a tracking system which measures the position and orientation of the TurtleBot2. In Unity, the absolute position of the robots base-link is taken from its transform and published to a corresponding ROS node. The complete procedure is repeated for a minimum of six times. The x-position of the robot at each stop is taken for every measurement. The mean value as well as the variance for the multiple iterations is calculated for each position. Those values are compared for the real robot and for the simulated robots.

4.4.8 UMBmark

A method for measuring and comparing errors in the odometry of mobile robots was already described in 1994 by Borenstein and Feng [47]. It is called UMBmark (University of Michigan Benchmark test). Despite of its age, it is still a widely used method within the mobile robotics community today. For the UMBmark the robot starts in a

known position close to a reference wall. It then drives a hardcoded, 4m x 4m square path in clockwise direction.

Under perfect circumstances, the position of the robot should be exactly the same as the starting position. Due to controller errors and "Dead-reckoning" errors, the robot can end up in a different position than the starting position. "Dead-reckoning" errors can be systematic or non-systematic, for example unequal wheel diameters or uneven floors, respectively. After finishing the square path, the absolute position of the robot is measured.

This procedure is repeated for five times in total. Afterwards, the same procedure is repeated for five times with a counter clockwise direction. Typical results of this test show two clusters of errors, one for the clockwise direction and one for the counter clockwise direction. The UMBmark is conducted in two angular directions, as certain errors can compensate each other. If that is the case, they would accumulate when the robot is running in the opposite direction.

The absolute end-position of each run is compared to the position that the robot is expected to have due to the odometry calculation. The odometry calculation is the position of the robot that is calculated through the velocity, v_{left} and v_{right}, and radii r of the wheels, as well as their distance d to the centre point. With those variables, the linear and the angular velocity of the robot can be computed. The calculation for a differential drive robot, like the TurtleBot2, can be done by using following formulas, according to [48]:

$$v_{\text{linear}} = {}^{r}/_{2} \cdot (v_{\text{left}} + v_{\text{right}})$$
$$v_{\text{angular}} = {}^{r}/_{d} \cdot (v_{\text{right}} - v_{\text{left}})$$

These velocities are then used with the timestep δt between two measurement points to compute the change in position and orientation of the robot. The frequencies for the measurements in this thesis are 50 Hz in Unity, approximately 30 Hz in reality and 500 Hz in Gazebo. The frequencies are calculated by dividing the duration of the rosbags with the amount of messages on the JointState topic. To simplify the calculations, the orientation of the robot is assumed to be constant during one timestep, which relates to maximum 0.33 s with the frequency of the real measurements. With this assumption, the position $[x, y]$ and orientation ω of the robot is calculated by:

$$x_i = x_{i-1} + v_{\text{linear}} \cdot \delta t \cdot cos\, \omega_{i-1}$$
$$y_i = y_{i-1} + v_{\text{linear}} \cdot \delta t \cdot sin\, \omega_{i-1}$$
$$\omega_i = \omega_{i-1} + v_{\text{angular}} \cdot \delta t$$

An example of ground-truth and calculated trajectories for an UMBmark is presented in Figure 8. The distance in x- and y-direction between the real endpoint and the calculated endpoint is computed for both directions and the points are presented in a scatter diagram. According to [47], these errors should, due to non-systematic errors, result in clusters around the systematic error of the system. This procedure in total is repeated for the real robot, the simulation in Unity and in Gazebo.

For this work, the UMBmark is adjusted. Instead of a 4 m x 4 m square path, only a 2 m x 2 m square path is tested. This deviation is necessary due to the comparison tests with the real robot. The absolute position of the real robot is tracked with a fix installed motion capture system in a laboratory. Even though the tracking area of this system is approximately 4 m x 6 m, the tested TurtleBot2 robot has too huge deviations in its trajectory to stay within the tracking area for the original, wider setup. As the tracking system has a mean error of less than 0.5 mm, it is seen as admirable to conduct

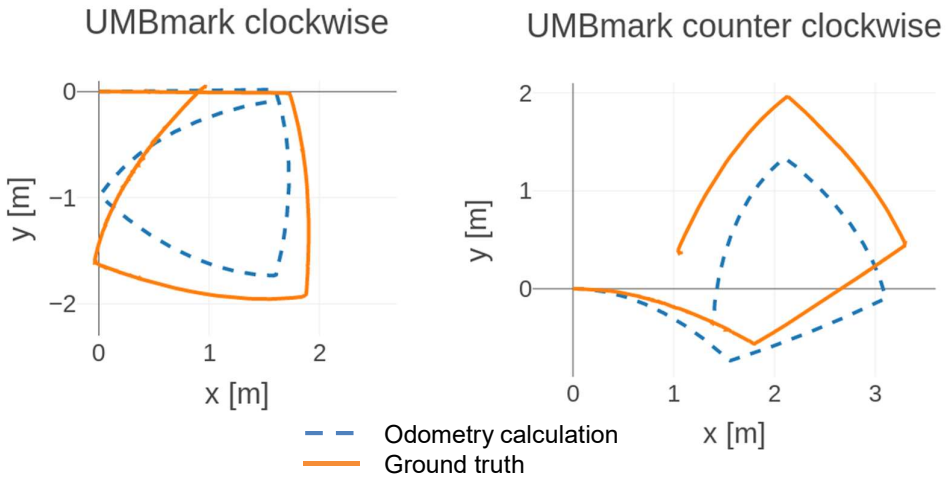


Figure 8. Absolute position and calculated odometry position in UMBmark for clockwise (left) and counter clockwise (right) direction.

the measurements within this area. The lengthwise reduction of the benchmark is not seen as an important disadvantage. Errors like different wheel diameters would have accumulated more with the original setup, as the robot is driving a longer way on each side. With the modified setup, the effect in general can still be observed, even if its extent is decreased in comparison to the original setup.

4.4.9 SLAM case study

To further investigate and evaluate the capabilities of Unity in combination with ROS, a SLAM (Simultaneous Localisation and Mapping) [49] case study is conducted. This case study is created to provide insight into the ability of Unity in combination with ROS to adapt to different algorithms and test scenarios. When working with mobile robots, SLAM is an often used technique that is used in the localisation and mapping process of the robot. The ability of Unity to connect to the corresponding algorithms is essential for its usage as a simulator and therefore an indicator for its applicability in robotics.

In this case study, it is attempted to use an open-source SLAM package with the simulation data of Unity. SLAM is needed, when a mobile robot neither has a map of its environment, nor does it have knowledge about its own pose in this environment. Instead, only measurements and control inputs are available. With those information, the algorithms have to acquire a map of the robots environment as well as locate the robots position in regard to that map. There are several sources for measurements that can be used in order to successfully apply SLAM algorithms. This can for example be laser scans, data from vision systems, from IMUs (inertial measurement unit) or from wheel encoders. In order to use Unity to simulate robots using SLAM algorithms, the sensory information need to be available in the simulation.

It is possible in Unity to render near-photorealistic imagery, as well as depth information, infrared or images with injected noise. In addition to that, LiDAR systems can be simulated via scripts [6]. There are example scripts available for simulating laser scanners in the Unity wiki [50] and simulating laser scanners with prewritten code for subscribing and publishing the corresponding messages in RosSharp [41]. Those scripts can then be attached to the corresponding laser scanner game objects in Unity to generate data. The data from IMUs can be simulated with scripts by using the differences

of the game object in position, in time and the gravity. There also exist example scripts for this purpose [51]. Data from wheel encoders can be used directly from Unity. An actuated normal wheel can be simulated with a HingeJoint in Unity. The properties of such HingeJoints include the current angle of the joint relative to its rest position in degrees and the angular velocity of the joint in degrees per second [26].

When it comes to the ROS community, tf messages are used to express the transforms between coordinate systems. These tf messages often are input and output of SLAM algorithms, as they specify the pose of the coordinate systems in regard to each other. An example of some of the nodes and topics in a SLAM setup can be seen in Figure 9. The rosbridge_websocket node is the node that publishes the messages from Unity. In this example, the messages that are published to the topics are the joint_states which inherit the name, position, velocity and effort of each joint and the scan. The scan topic includes the laser scan data from the laser scanner. Here, the range data as well as the scanned angles, the scanned time and the minimum and maximum ranges are included.

The joint_states are subscribed to by the robot_state_publisher node. This node calculates the tf tree of all the transforms of the robot. To do that, it uses the data from the joint_states and the URDF file that need to be specified when launching the node. Within the tf tree, there are two types of messages. One is called tf_static and includes the static transforms which does not change during the simulation, as they are not dependent on any variables. The second type of messages are the variable transforms that change because of moving joints. Those messages are called tf.

The last node in this example for using a SLAM algorithm is called slam_gmapping. It is created from an open source package called gmapping [52]. It includes a SLAM algorithm and subscribes to the scan and all transforms. It internally uses all the information to update the pose of the robot by adjusting its transform and creates as well as modifies the map. This map then, in combination with the transforms, can be used in graphical displays of the robots actual position within its environment.

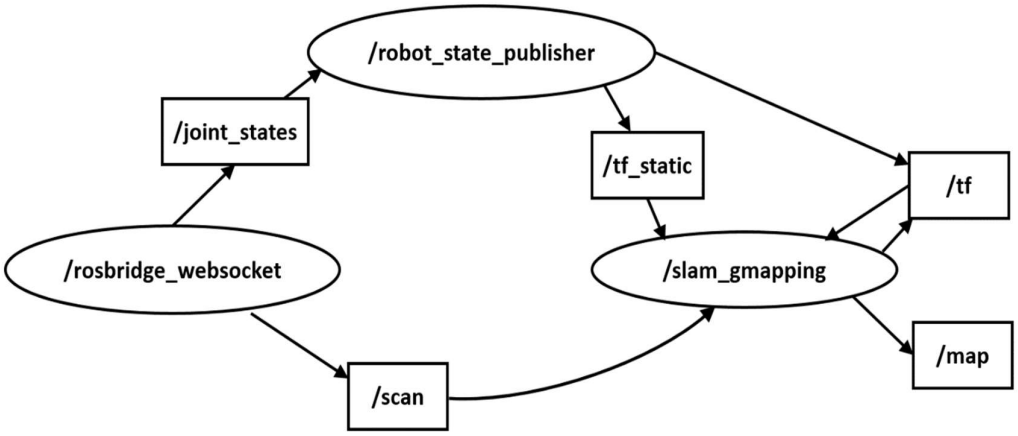


Figure 9. Active nodes for SLAM algorithms.

5 Results and discussion

In this chapter, the results of the conducted investigations and measurements are presented and discussed. For all benchmarks a friction value of 1.4 in Unity is used, if not stated otherwise. This is due to comparisons of the trajectory of the UMBmark and is explained more in detail in Section 4.4.8. For the Gazebo experiments, the friction parameters mu1 and mu2 are set to 1.0 if not stated otherwise. These are the default friction values for Gazebo when running with ODE. All simulations in Gazebo are conducted by employees of the Robert Bosch GmbH. The data is evaluated and visualized by the author of this thesis work.

This chapter is structured into sections for each benchmark. The first three benchmarks are the stacking test, the collision test and the integrator test. Those are basic benchmarks which should provide an insight of Unity's capacities as well as the influence of certain parameters on the simulation results. Section 5.4, 5.5 and 5.6 present and discuss the results of the μ-split, ramp and friction turn benchmark. Those are benchmarks that are evaluated in a qualitative way, namely a comparison between the behaviour of the real and simulated robots for different circumstances. In Section 5.7 and 5.8, the results of the basic drive benchmark and the UMBmark are presented and discussed. Those benchmarks are evaluated in a quantitative way. The last section presents and discusses the results of the SLAM case study.

5.1 Stacking test

The stacking test was presented in Section 4.4.1. The timestep has been varied from 0.001 s to 0.03 s in steps of 0.001s. The amount of boxes which resulted in a collapsing stack can be seen in Figure 10. The blue solid line represents the default settings, while

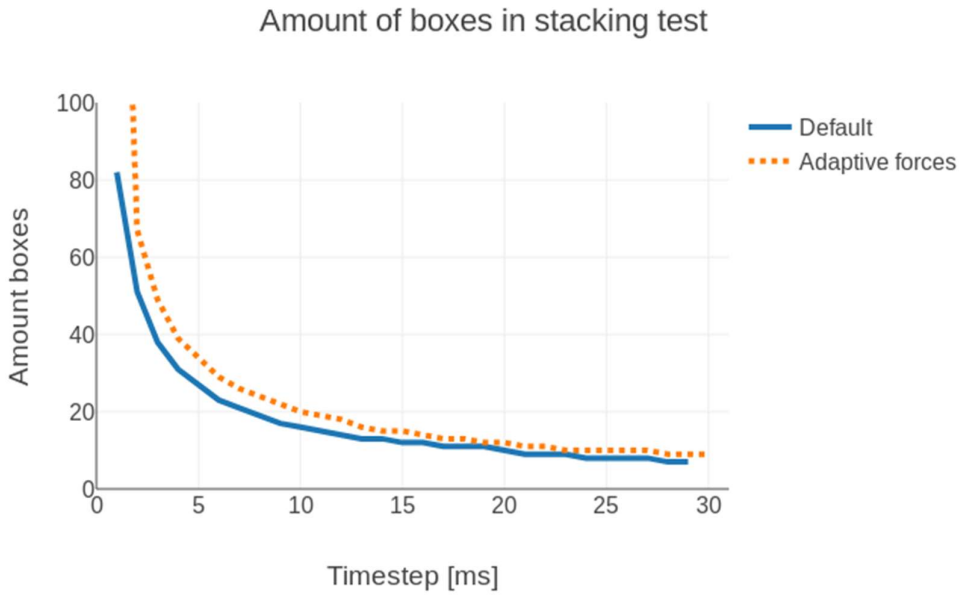


Figure 10. Amount of boxes that led to a collapsing stack in the stacking test.

the orange dotted line shows the results with the same settings, but with enabled adaptive forces. Adaptive forces and their influence on the simulation have been discussed in Section 4.1.

In general, a smaller timestep led to a higher amount of boxes that could be successfully stacked onto each other. If the timestep got bigger, the stack collapsed more easily. This is reasonable, as a smaller timestep led to more calculation iterations and therefore a smaller numerical error in the computation. The numerical error in the calculations led to a drift in position and orientation of the boxes and therefore their collapse.

If the adaptive force was enabled, more boxes could be stacked onto each other. This effect was especially noticeable for small time steps and vanished with increasing time steps. This seems to be reasonable, as the adaptive force should give more realistic behaviour for the transmitted forces of a stack or pile of objects, as described in the Unity manual [26].

The behaviour of the stack in Unity was significantly different than the behaviour in previous physics engine evaluations and Gazebo. In Unity, the boxes stuck on each other rather consequently and only the whole stack itself started to move or oscillate, respectively. This behaviour can be seen in Figure 11 on the right side. The boxes here did not move against each other, they look like they were glued together. In the contrary for the same test in Gazebo, each box had a little offset in regard to its upper and lower neighbours. This can be seen in Figure 11 on the left hand side. The same behaviour could be seen in a picture of the stacked body stability test in [38]. Here, the physics engines Bullet, ODE and PhysX were compared with each other, but it was not stated which engine was used for the simulation that was shown in the picture of the stacked body stability test.

This difference in movement led to differences in the collapsing. For Unity, the stack oscillated until it broke at some point and the upper half of the stack began to fall down to the ground. For Gazebo, the small movements accumulated. The stack collapsed because of the internal position mismatches and gravity that forced some boxes to fall down towards the ground. As this behaviour occurred because of numerical errors in the physics calculations, it could not be rated what the behaviour of the collapsing stack should be like. It seemed to be a difference in the physics engines ODE, which was used in Gazebo, and PhysX, which was used in Unity.

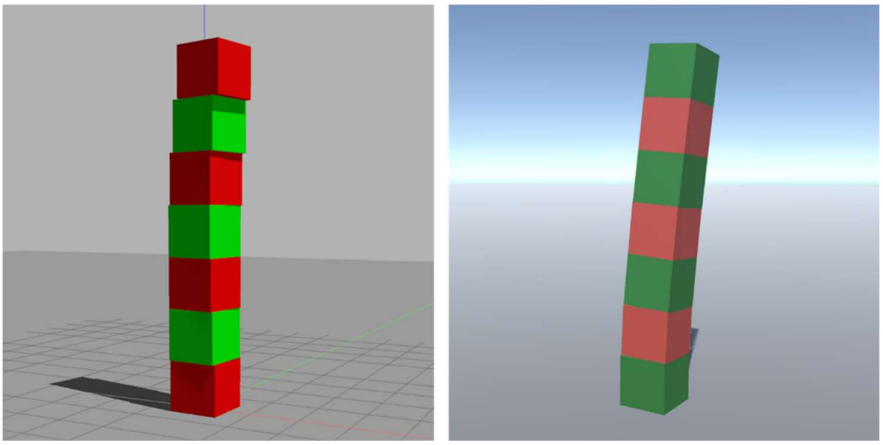


Figure 11. Different behaviour of stack of boxes in Gazebo (left) and Unity (right).

5.2 Collision test

The second basic benchmark was the collision test, as explained in 4.4.2. Spheres were positioned above a top-down, four sided pyramid. With the start of the simulation, the spheres were accelerated towards the pyramid through gravity and collided into each other and the pyramid. The maximal penetration of each time step between all spheres and the pyramid in one simulation can be seen in Figure 12.

The simulation was repeated for fixed time steps from 0.006 s until 0.030 s in increments of 0.002 s. It can be seen that an increased fixed timestep led to an increased maximal penetration for the simulation. This seemed to be reasonable, as the collision detection was executed less regularly. When the collision detection was executed less regularly, the objects could move further with the same speed between two calculations. This led to increased penetrations, as objects that have just not been detected in the previous step could travel further into each other before the next calculation was executed.

The distribution for all time steps had their minimum at 0.00 m, as the calculation of the maximal penetration started with the beginning of the simulation. The objects were positioned in the beginning so that they didn't touch each other. Nevertheless, the distributions showed that the mean and maximum penetration for each simulation rose with an increasing timestep. This needed to be considered especially if a simulation handled objects with high speeds. If that was the case and the timestep was chosen too big, it could happen that collisions were not detected at all or the penetration of two objects was comparably high. If a reliable collision detection is needed with high-speed objects, special care is needed when setting the fixed timestep of the simulation.

5.3 Integrator test

The third and last basic benchmark was the integrator test. A sphere was accelerated through gravity for a defined amount of time. After that time, the absolute position of the sphere was compared to the position that the sphere should have according to theory. No drag was taken into account. The benchmark was explained more in detail in

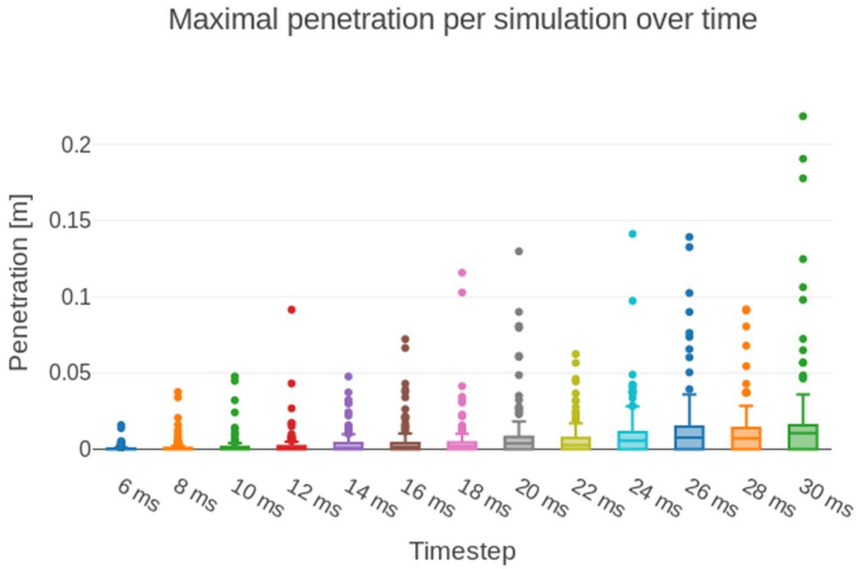


Figure 12. Maximal penetration in the collision test.

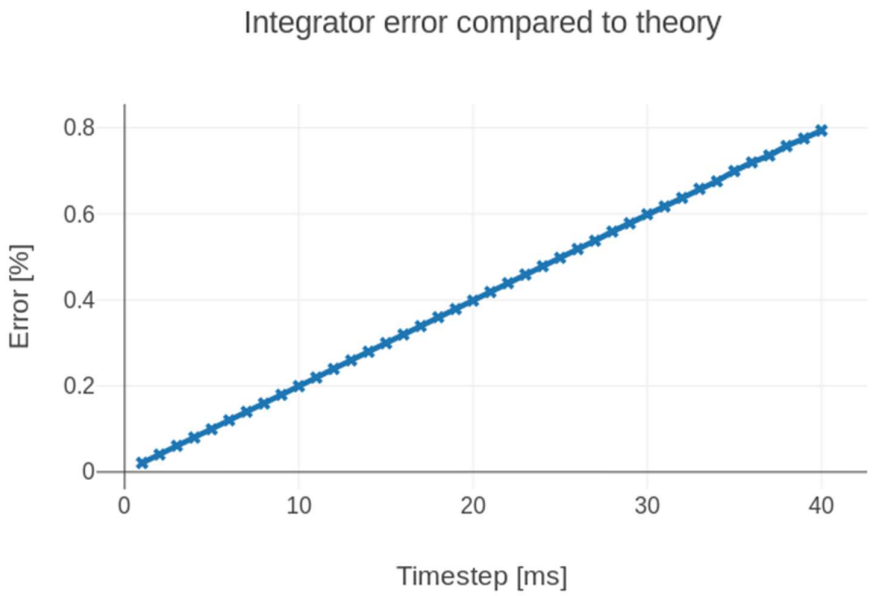


Figure 13. Integrator test with a sphere as described in Section 4.4.3.

Section 4.4.3. The results of this benchmark can be seen in Figure 13. The error of the simulation was calculated in regard to theory and plotted against the fixed timestep of the simulation. The error was proportional to the timestep, i.e. the larger the timestep, the less accurate the velocity integration of the physics engine.

This could be confirmed with theory, as the amount of calculations in the physics engine would decrease with increasing time step and therefore a longer period of time between two calculations. If the velocity was calculated less regularly, the error would increase as changes in velocity due to accelerations would be registered and applied to the object less frequently.

5.4 μ-split

The first of the qualitative benchmarks was the μ-split. The TurtleBot2 was placed on a floor which was split into two areas with different friction values. The TurtleBot2 was initially placed so that the wheels were experiencing different friction values. Initially, the left wheel was in contact with the higher friction value and the right wheel with the lower friction value. One of the friction values remained constant at the chosen default value of 1.4, while the other value was variable and increased with each iteration. The robot was expected to turn towards the lower friction side due to slipping when the friction values there became too low.

The benchmark has been described in Section 4.4.4 and the results of Unity can be seen in Figure 14. The stated friction values have been set for both dynamic and static friction. It can be seen that, in the tested lower segment of friction values between 0.00 and 0.40, a decreasing friction coefficient led to more rotation of the robot and therefore more curvature of its trajectory under otherwise identical circumstances. This was probably because of more slipping of the lower-friction-wheel and therefore a less equal force-distribution on the robot.

The unequal force that the wheels exceeded on the robot led to an angular rotation around the z-axis of the robot. This rotation of the robot was changing its driving direction to not be parallel to the border between the floors anymore. This led to the

curvature in the trajectory that can be seen in Figure 14. Because of this curvature the left wheel that was placed on the higher friction floor before at some point was driving across the border between both floors. After crossing the border between both floors, it was in contact with the lower friction ground, like the right wheel.

When taking a closer look to the trajectory, there exist differences between the different friction values. For the lowest friction value of 0.00, the angular momentum was the highest. This could be seen as the turning radius for 0.00 was smaller compared to the turning radius of the higher friction values in Figure 14.

Because of this angular momentum, the robot rotated around its own z-axis as soon as both wheels were in contact with the lower friction floor. As both wheels were now experiencing the friction value of 0.00 towards the ground, the angular rotation of the robot would not result in a difference of the movement of the robot. The robot moved like a rotating, flat object on ice. The rotation of the robot did not influence the trajectory, but the direction of movement that the object had before entering the ice was still exceeded, as it was not limited by friction forces. This can be seen in the straight line for the trajectory of the 0.00 friction value after both wheels have entered the lower friction area. For a friction value of 0.02, the angular rotation was affecting the movement of the robot even after both wheels were in contact with the lower friction floor. The trajectory for the friction value of 0.02 was changing direction significantly after the wheels have been in contact with the lower friction side for approximately one meter. There could not be found an easy straightforward explanation for this behaviour. With increasing friction, the resulting angular momentum on the robot was decreased, as can be seen with the bigger turning radii. The trajectory of the friction values higher than 0.02 show mostly no change of direction after both wheels are in contact with the lower friction side.

The measurement with a friction value of 0.00 was significantly shorter than the other trajectories. This could be explained with the fact that the rotation took part rather

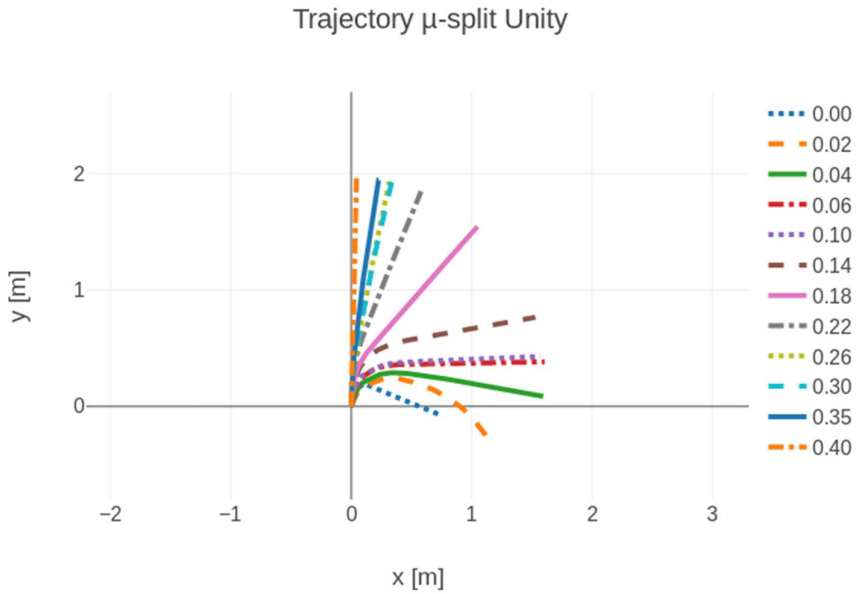


Figure 14. Trajectory of the μ-split scenario in Unity. The TurtleBot2 model is placed on [0, 0] in the world coordinate system with its base link at the start. For x≤0, the plane has the default friction value of 1.4. For x>0, the plane has different friction values for each run which are indicated in the legend.

fast due to the high angular momentum. The TurtleBot2 therefore only experienced the acceleration through the left wheel for a short amount of time and did not reach a high speed. As soon as both wheels were in contact with the slippery ground, no acceleration at all was transferred anymore. A friction value of 0.00 was described to behave like ice in Unity's manual [26].

The results of the μ-split scenario in Gazebo can be seen in Figure 15. The basic behaviour of the simulated robot in Gazebo was similar to the simulated robot in Unity. Both scenarios showed a rotation towards the lower friction side. The friction value of 0.00 was the lowest value that could be set in both simulators. The friction value in Unity could be set for a range of $0 \leq x < \infty$ [26], while the friction values in Gazebo could be set in a range of $0 \leq x \leq 1$ [53]. Note that according to [54], Gazebo would choose the minimum specified friction value of two objects for the friction combination. The friction value of the wheels in Gazebo was set to 1.0, according to the URDF [46], so the resulting friction values were the ones of the lower friction side when the wheel was colliding with that floor. The rotation in the trajectory of the Unity simulation for 0.00 was significantly more distinct than in Gazebo. In Unity, the TurtleBot2 rotated its direction of movement around approximately 120°, while the rotation in Gazebo was slightly below 90°. For both simulators, the rotation towards the lower friction side decreased with rising friction values for that side.

The trajectory for the higher friction values showed less rotation in the Gazebo simulation than in the Unity simulation. This can be seen in Figure 15, where the trajectories were showing significantly less curvature than those in Figure 14. The friction values are not used for computations in the same way for ODE and PhysX. In PhysX, the friction values are divided in dynamic and static friction. In ODE on the other hand,

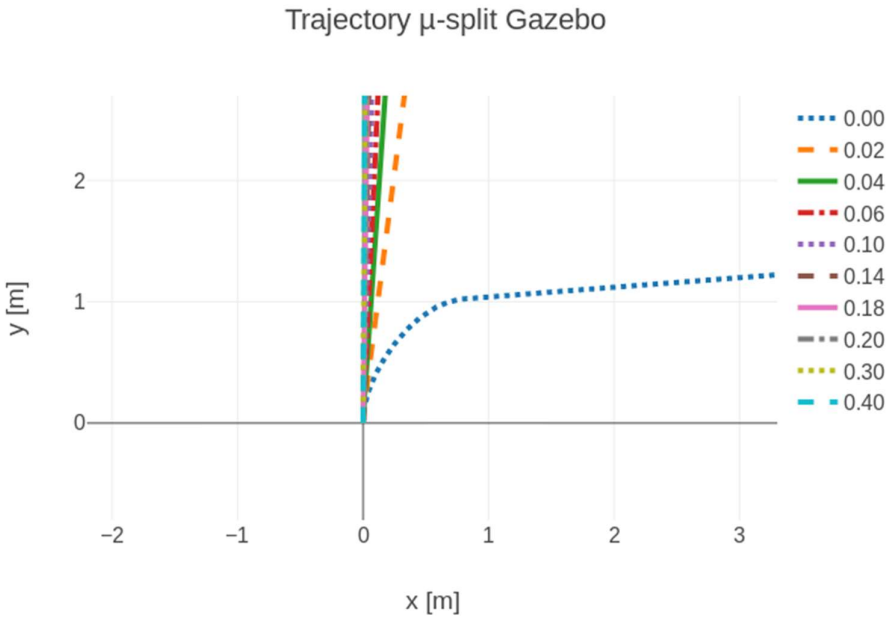


Figure 15. Trajectory of the μ-split scenario in Gazebo. The TurtleBot2 model is placed on [0, 0] in the world coordinate system with its base link at the start. For x≤0, the plane has Gazebo's default friction value of 1.0 for mu1 and mu2. For x>0, the plane has different friction values in mu1 and mu2 for each run. Those values are indicated in the legend.

the two friction values describe the Coulomb friction coefficients for the first and second friction direction. The effect of the different models was also described in an experiment of a physics evaluation research [38]. Because of this, the trajectories for the same input friction values were not expected to be the same for simulations in Gazebo and Unity. Therefore, no conclusion could be drawn of the difference in curvature for those friction values in Gazebo in Unity. Nevertheless, both simulators showed the expected qualitative curvature that was induced by the slipping of the wheel on the lower friction side. Note that both robot simulation models as well as the real robot were given input control messages to drive with a speed of $0.2\ {}^{m}/_{s}$.

For the real robot, this scenario was conducted by sticking tape on one of the wheels. When actuating both wheels with the same momentum, the wheels of the robot did not slip and cause a measureable rotation of the robot similar to the experiments in the simulation. For further decreasing the friction value, tape was also stuck on the floor. The TurtleBot2 was placed so that the wheel with the tape was in contact with the tape on the ground floor. Still, no rotation or change in direction could be observed. Under closer observation, the wheel with the tape did not seem to slip during the motion. This explained the not-existent rotation or change in direction for this benchmark in reality.

The real TurtleBot2 was expected to have a similar movement to the Unity simulation if the friction on one wheel got small enough to induce the wheel to slip. Note that the friction values that result in the significant rotations in the simulations were significant lower than the default values. The highest friction value that resulted in a notable curvature of the robot's trajectory in Unity was 0.35, while the default friction value for benchmarks in Unity was 1.4. With that friction value, the trajectory of the UMBmark in Unity was most similar to that of the real robot, see Section 4.4.8. As can be seen in Figure 14, only friction values lower than 0.40 for the lower friction side resulted in a significant curvature. The endpoint of the trajectory for the friction coefficient of 0.40 was at $[0.04, 1.96]$. It was not possible in the extent of this thesis to reproduce this behaviour with the real robot. It was assumed that the friction values were too high. To further decrease the friction, the surface of the wheels would probably need to be adjusted or the TurtleBot2 would need to be put on wet surfaces or ice.

5.5 Ramp

The next qualitative benchmark was called ramp. A TurtleBot2 was placed on a ramp with a variable slope and its trajectory was compared for different slopes. The robot was controlled with the same script for each iteration, so that the trajectory should be identical for several simulations under the same circumstances. The x and y position of the robot was measured relative to the ramp. This was chosen so that the same travelling distance on the ramp would lead to the same x and y position for the trajectory of the robot. The difference in height was taken into account when referring to the ramp position. This benchmark was described in Section 4.4.5. Experiments with the ramp benchmark were done with the real robot, Unity and Gazebo.

The results of Unity can be seen in Figure 16. With an increasing slope, the travelling distance of the TurtleBot2 decreased for the same control inputs. This can be seen in Figure 16, as the trajectory got shorter for increasing slopes. In addition to that, the TurtleBot2 started to move sideways significantly at a slope of approximately 30°. When that happened, the travelling distance of the TurtleBot2 increased again, as the sideways movement led to a decreased slope that needed to be travelled by the robot.

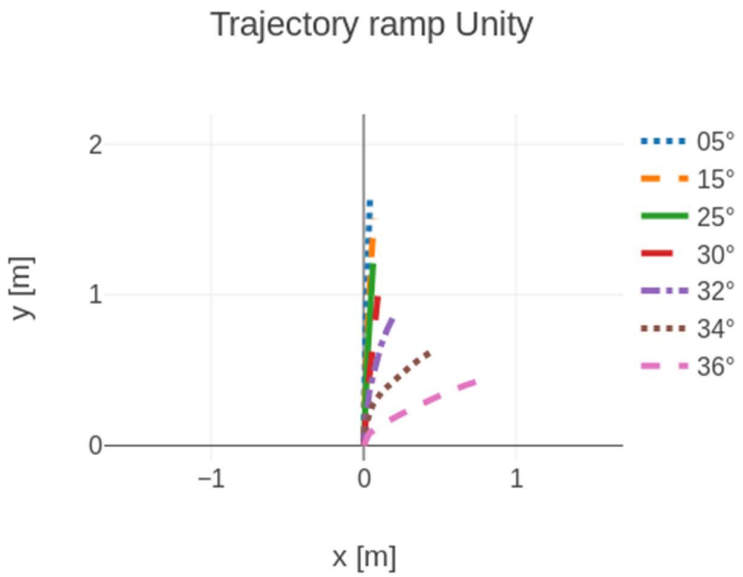


Figure 16. Trajectory for ramp benchmark in Unity.

The sideways movement could also be observed for the measurements with the real robot. When the slope got too high, the TurtleBot2 started to turn slightly and therefore drove sideways instead of straight up the ramp. With the real TurtleBot2, there could not be found a slope where the robot could not drive straight up the ramp anymore. Instead of that, the robot started to move sideways, similar to the movement in the simulation.

If the slope of the ramp was further increased and the robot was prevented from moving sideways, it started to rotate backwards around its transverse axis and therefore tip over itself and fall down the ramp. This incident probably occurred due to two reasons: firstly, the wheels of the TurtleBot2 had a comparably high friction coefficient and stick to the ground really well. Because of this, the robot could climb up the ramp

Figure 17. Picture series of how the robot tips backwards over itself. The left picture shows the robot in its starting position. In the right picture, the wheels of the robot are actuated and the robot starts to tip over itself.

and leave its starting position when the ramp had a comparably high slope of 30°. Consequently, the robot was rotated 30° around its transverse axis in regard to the floor. A slope of 30° related to an increase of height of 0.5 m for a 1 m drive in x-direction.

Secondly, the spring hanger of the wheels is designed in a way that it exceeds a momentum around the transverse axis of the robot. The springs are implemented in the Kobuki base to ensure contact of the actuated wheels with the ground. In that way, the springs press the actuated wheels on the ground. The resulting momentum of the springs' exceed force around the transverse axis of the robot. This momentum seemed to increase when the wheels were accelerated. In sum this led to a higher original transverse rotation of the robot, due to the high slope of the ramp, and an extra added force, due to the spring hanger of the wheels. The combination of both reasons probably led to the incident that the robot tipped backwards over itself.

A picture series of this incident can be seen in Figure 17. The left picture shows the TurtleBot2 in its starting position. It was prevented from rolling down the slope backwards by an obstacle. The wheels of the TurtleBot2 were accelerated. This led to a forward movement combined with a "rearing up" movement. In this progress, the TurtleBot2 tipped over itself, depending on the slope of the ramp and the acceleration. This behaviour could not be replicated with Unity, as the spring hanger of the wheels was not modelled in the TurtleBot2 URDF, as explained in Section 4.3.

The results of the ramp benchmark in Gazebo can be seen in Figure 18. The experiments were conducted with the same slopes like the Unity experiments. The trajectory showed no significant difference for the increasingly high slopes. In contrast to the simulation in Unity and the experiments in reality, no sideways movement of the robot could be observed. The length of the trajectory between the different slopes seemed to vary slightly. Upon closer observation, the difference in the length of the trajectory varied for 0.2 m between the slope of 5° and the slope of 36°. This corresponded to a difference of 6% in the length of the trajectory and was significantly lower than the changes in Unity. Hence, no significant difference in the trajectory could be found for the Gazebo simulations.

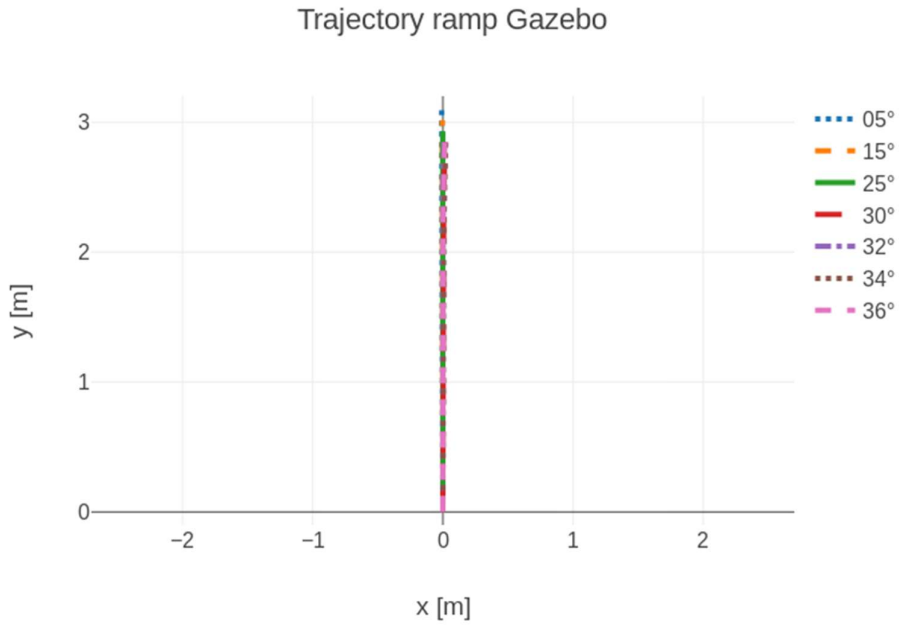


Figure 18. Trajectory for ramp benchmark in Gazebo.

When taking a closer look to the differences in the simulation of Unity and Gazebo, this incident could be explained. In Unity, the wheels were simulated with a hinge joint that was able to be defined using a motor. This motor then used a in the manual not further defined controller for computing the actual velocity in regard to the input target velocity. [26] In contrast, the default actuation of the wheels in Gazebo did not inherit any kind of controller. Instead, the target velocity was instantly set as actual velocity of the wheels [55]. This meant that, as soon as the robot was ordered to drive with a certain velocity, this velocity was realized of the wheels. The start-up of the motors and the wheels which needed to overcome the inertia of the system when starting to rotate was not modelled. This also meant that the effort of the wheels for rotating was not taken into account when setting their actual velocity.

When this was the case, the increasing slope of the ramp should not have an effect on the trajectory of the robot. The difference in trajectory and the increased sideways movement most probably evolved from the increased effort it took the robot to climb up a ramp with a higher slope. With the non-existent controller of the Gazebo simulation, the effort was not taken into account when setting the velocity of the wheels. Because of that, the increased effort for higher slopes should not affect the movement of the robot. To get more realistic results for the Gazebo simulations, a controller should be implemented. If for example a PID controller for the velocity of the wheels would be implemented in the simulation, the results might match the results of Unity and reality more closely.

In addition to the mismatch in the controlling, the friction parameters of Unity and Gazebo were not directly comparable. The implementations for friction parameters in Unity and Gazebo were different. Because of that, it could be that the same friction values would correspond to different behaviour on the ramp even though all other settings might be identical. It has therefore to be taken into account that the results were not comparable in a quantitative but in a qualitative way, hence the qualitative behaviour of the robot instead of the absolute positions in its trajectory.

5.6 Friction turn

The third and last qualitative benchmark was called friction turn. It has been described in Section 4.4.6. The evaluation was to observe, if a collision with an object in certain degrees will lead to a turn of the robot. The results can be seen in Figure 19. Picture a shows the robot 0.1 s after the start of the simulation. The TurtleBot2 had just started to drive and the angle between its y-axis and the cube was 20°. In b, the TurtleBot2 has been driving for 1.3 s. It was close to reaching the cube and still had the original angle of 20°. In c, the simulation has been running for 2.0 s. The robot has hit the cube and turned 4° towards it already, as it had an angle of 24°. When continuing the simulation, the robot would turn even more towards the cube, as can be seen in Picture d. The simulation has been running for 2.9 s in total at that point and the TurtleBot2 turned 24° towards the cube since hitting the cube for the first time.

This correlated to the expected behaviour of the mobile robot. When continuing to drive forward after hitting an object in that way, the friction between the object and the robot would, provided that the object was heavy and steady enough to not just be pushed away by the robot, result in a turning movement. This behaviour has been confirmed with measurements of the real robot. The results of these measurements were a qualitative observation of the behaviour of the robot. The TurtleBot2 was driven against objects in an angle which was approximately similar to the 20° of the simulation.

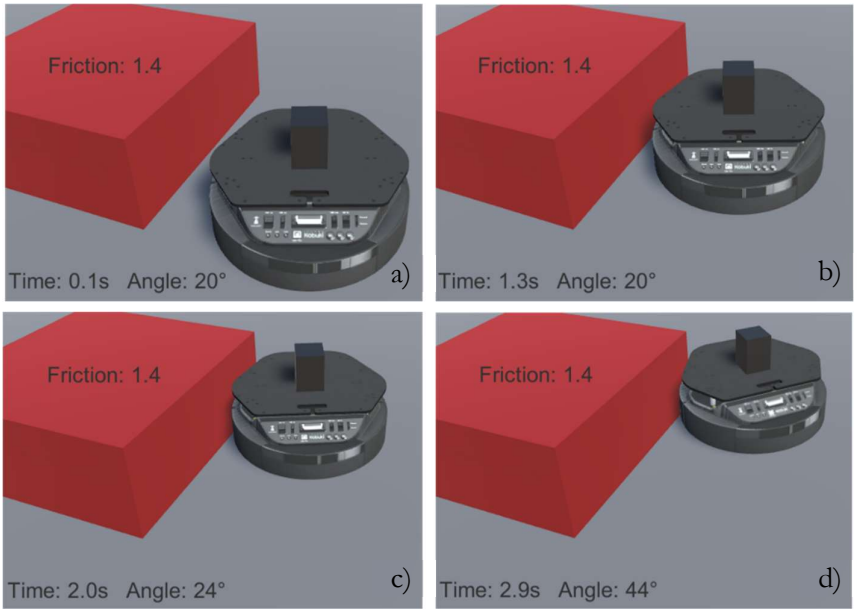


Figure 19. Friction turn benchmark in Unity. The picture series shows the approaching and then turning robot at different times.

The behaviour of the real robot during this benchmark test in reality can be seen in Figure 20. The robot was actuated to drive forward. It started to turn towards the object as soon as colliding with it. The robot continued to turn towards the object if the turning of the wheels was still resulting in a forward force.

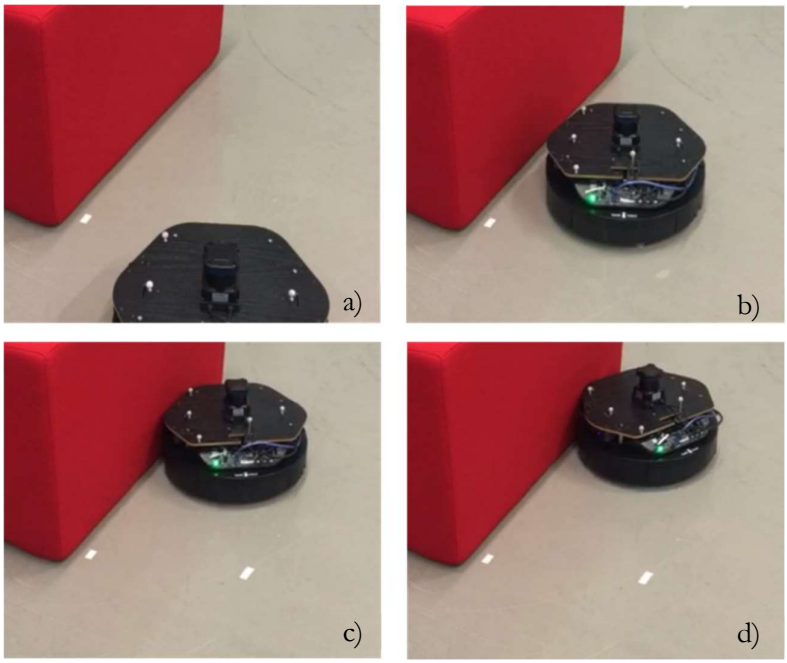


Figure 20. Friction turn benchmark with the real robot. The starting position of the TurtleBot2 can be seen in a. Picture b shows the pose of the robot directly before the collision with the object. Two stages of rotation can be seen in c and d.

5.7 Basic drive

Two quantitative benchmarks were conducted in this work. One of them was called the basic drive benchmark. The TurtleBot2 robot was controlled to drive forward and backward with different speeds and its absolute position after each stop was measured. The implementation was already described in Section 4.4.7. The measurements were conducted in reality, in Unity and in Gazebo. Each of those measurements was repeated for at least six times. The results can be seen in Figure 21.

The control script gave the TurtleBot2 input to drive $\pm 0.2\ ^{m}/_{s}$, $\pm 0.25\ ^{m}/_{s}$ and $\pm 0.3\ ^{m}/_{s}$ for 20s each. With a perfect controller, a perfect odometry of the robot and no inertia, this would lead to the following theoretical positions for the robot. As no angular velocity was given for the robot at any point of time during the simulation, all y-positions of the robot should be zero. The first position of the robot should be at $0.2\ ^{m}/_{s} \cdot 20\ s = 4\ m$. As the same speed for the same duration was applied in negative direction after each forward drive, the second, fourth and sixth position should be at the starting point of 0m. The third position of the robot should be at $0.25\ ^{m}/_{s} \cdot 20\ s = 5\ m$ and the fifth position at $0.3\ ^{m}/_{s} \cdot 20\ s = 6\ m$.

As can be seen in Figure 21, neither the real robot nor the simulation in Unity reached the theoretical end positions. This probably happened out of two main reasons. The first reason were the already mentioned physical effects. The controller of the robot was not perfect and out of physical reasons like the inertia of the robot it took time until the target velocity of the robot was reached. The second reason was the rotation of the robot. An example of the trajectory in reality, Unity and Gazebo for this scenario can be seen in Figure 22. Even though no angular velocity was given, the robot rotated to a certain extent in reality and Unity. Therefore it did not only drive in x-direction, but also in y-direction.

This could happen out of many reasons like different wheel diameters, the kinematic characteristics or differences in the motor performance. The degree of this rotation and therefore the movement of the robot in y-direction was significantly higher in the Unity simulation than in reality. For example, the fifth position in reality was at approximately [5.5, 0.3], while the TurtleBot2 in Unity is at [5, 1.5] at that time. This significant amount

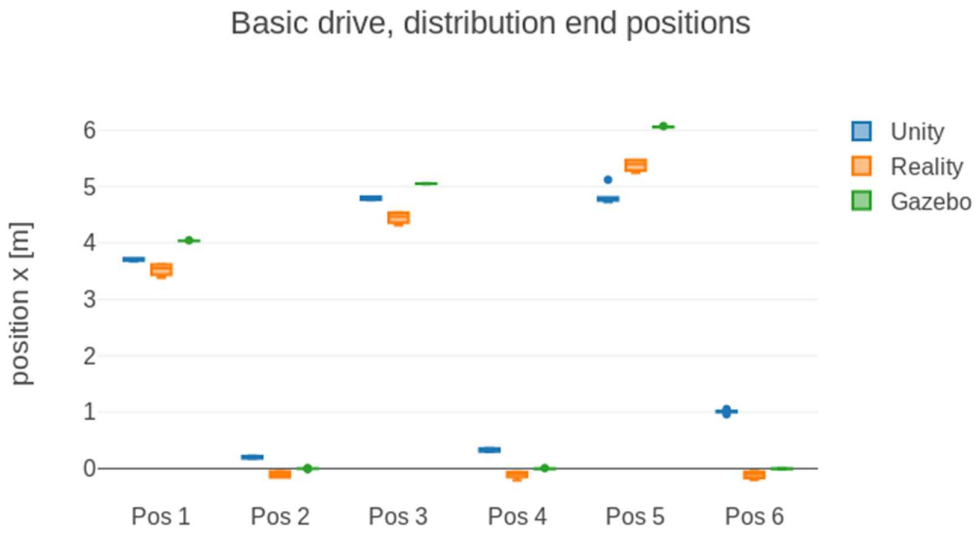


Figure 21. Basic drive evaluation for reality, Unity and Gazebo.

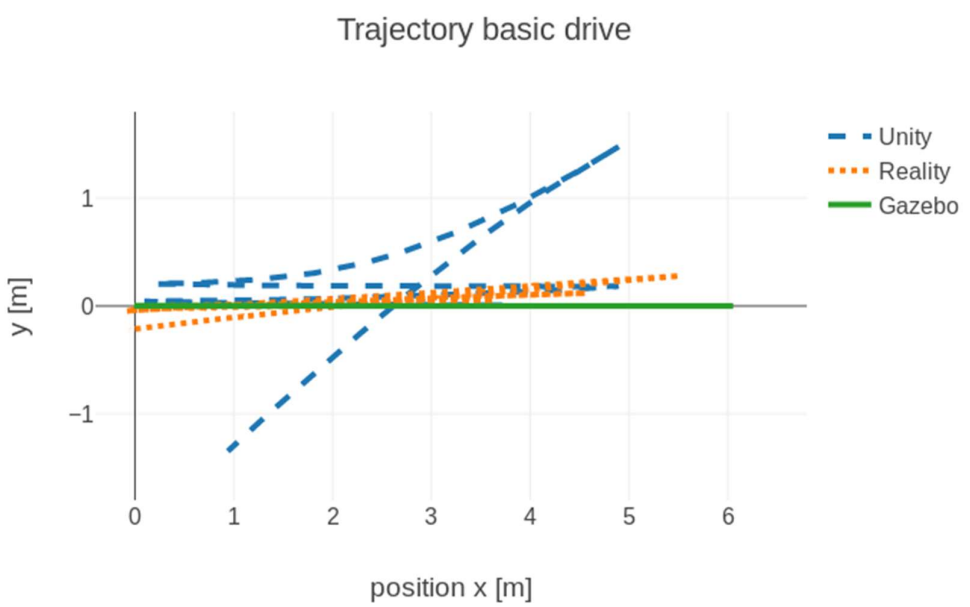


Figure 22. Example of the trajectory in the basic drive scenario.

of curvature the trajectory of simulations in Unity could only be observed for the higher speed of $0.3\ ^{m}/_{s}$. With lower speed, the curvature of the trajectory in Unity was comparable to the one in reality. The trajectory curvature in reality did not show a similar change of degree with rising speed.

This curvature could not be seen in the trajectory of the robot in Gazebo. The TurtleBot2 here drove a perfect straight line and did not change its driving direction at any time. This correlated to the finding in Figure 21. The robot in the Gazebo simulation was the only robot which did end up at the theoretical positions with high precision. As for the proximity to the theoretical positions, the simulation in Gazebo was the only experiment where no controller for the velocity of the wheels was used. This incident already had effects on the results of the ramp benchmark in Section 5.5. In this case, the direct usage of the velocity input on the wheels and the consequent violation of inertia in the whole system led to reaching the theoretical positions with the robot in the Gazebo simulation. There could not be found a cause to why the simulation in Gazebo did not show a curvature of the trajectory while the simulation in Unity and the experiments in reality did.

5.8 UMBmark

The second quantitative benchmark was based on the UMBmark [47] and described in Section 4.4.8. When first conducting the measurements in Unity with the default physics parameters and therefore friction values of 0.6, the trajectory was significantly different to the measurements with the real robot, as can be seen in Figure 23. The trajectory in reality was represented by the blue, solid line while the trajectory of the Unity simulation was shown with the dotted, orange line. For the simulation, the rotations in each corner were only approximately 30° instead of the hard-coded 90° in the control script. This led to an immense difference for the trajectory, as the movement after each corner was directed into a totally different direction. Even though the rotation would probably not reach 90° due to the used controller in both reality and simulation and the

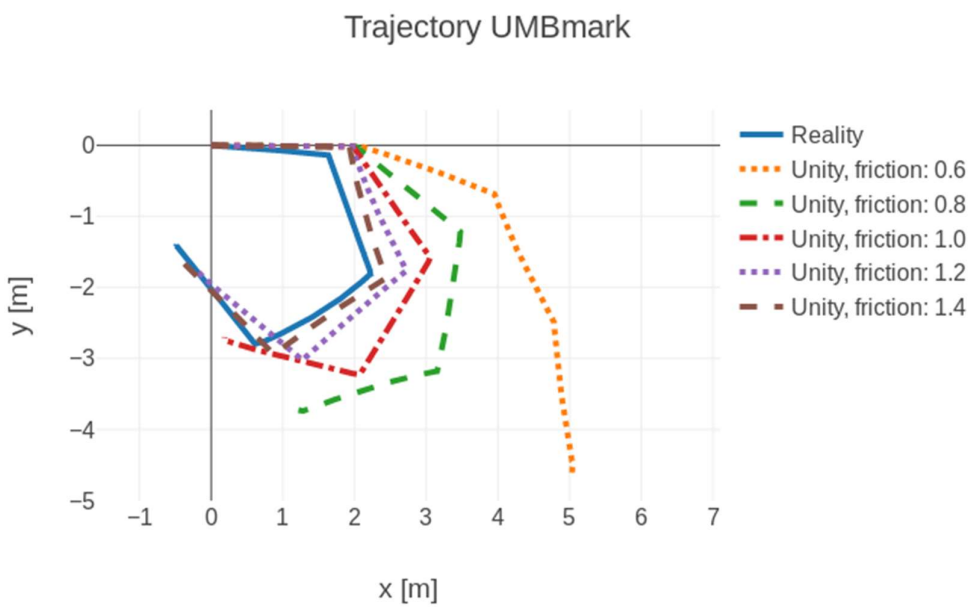


Figure 23. Trajectory UMBmark with different friction values for the wheels in Unity.

inertia of the robot, only 30° seemed to be too little, especially since the rotations of the real robot were approximately 70°.

Further investigation of the JointState message was conducted. This message contains the position, angular velocity and effort of a joint. The angular velocity of the wheels at each timestep was used to compute the linear and angular velocity of the robot at that time. This was called odometry measurements and the used formulas for the calculation have already been stated in Section 4.4.8. The values for the angular velocity of the joints during the scenario were integrated over time for each rotation. The resulting values showed that the theoretical rotation of the robot should be approximately 80°. These values were significantly higher than the rotation that can be seen in the trajectory for a friction value of 0.6 in Unity, as shown in Figure 23. If the theoretical rotation of the wheels was fitting to the feedback of the sensors via the JointState message, as shown through the odometry measurements, the mismatch in rotation did have to come from a different source.

The new assumption was the wheels rotated correctly, but slipped on the ground from time to time when turning or starting to drive. Therefore the robot didn't experience the full force through the turning of the wheels and moved less in regard to the theoretical trajectory. To further investigate this possibility, the UMBmark measurements were repeated with different friction values for the wheels of the simulated TurtleBot2 in Unity. As can be seen in Figure 23, the trajectory got more similar to the real trajectory with higher friction values. The most similar trajectory of the Unity simulation in regard to the measurements in reality was with the friction value of 1.4. This trajectory can be seen in Figure 23 with the dashed, brown line.

The friction values of the real wheels against the concrete in the laboratory were not measured for this thesis. In addition to that, the friction values in reality were not assumed to be constant due to small amounts of dust and other particles in the laboratory. Because of this and as the trajectory with friction values of 1.4 was most similar to the trajectory of the real robot, these have been used as standard friction values for the Unity measurements. With those friction values, ten iterations of the UMBmark have been conducted in Unity.

A problem occurred when taking a closer look at the results of the real robot with the motion capture system. As the robot was placed in the starting position manually for each run, it was not at the exact pose for each run. The position could be easily set to zero for the results by subtracting each entity of the x- and y-position with the first measured x- and y-position. The orientation of the robot for the beginning position was not as simple. When simply using the first measured orientation and then rotating the x- and y-positions around the origin according to that orientation, the graphs were not starting in the same directions but had angular offsets up to 30°.

Instead of using the orientation that was recorded by the motion capture system, the position of the robot was used to calculate the direction it was heading towards. To do that, the trajectory was first set to begin at the origin, as already mentioned above. Then, a program looped through the list of x- and y-positions until one of these positions became greater than the threshold of 0.1 m. This was done in this way instead of taking a fixed entity of the measurements to ensure that the robot was really driving towards that direction and not only experiencing a greater delay between starting the measurements and starting the control script. The x- and y-positions of the entity that exceeded the threshold were used to calculate the orientation α by:

$$\propto = tan^{-1} {}^{y}/_{x}$$

This angle α was afterwards used when again looping through all entities of the x- and y-positions to rotate each point of the trajectory according to the orientation of the graph. This was done with the following equations:

$$x = x \cdot cos\,\alpha - y \cdot sin\,\alpha$$
$$y = x \cdot sin\,\alpha + y \cdot cos\,\alpha$$

The resulting positions were used to plot the trajectory and calculate the errors between the ground truth and odometry end positions. These calculations only had to be done

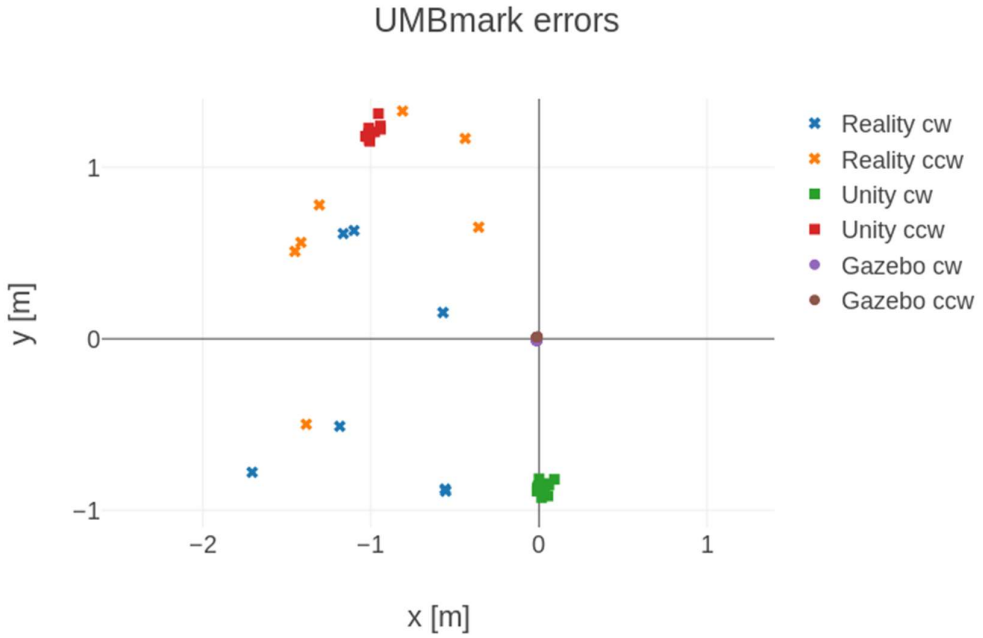


Figure 24. Errors between the odometry position and the ground truth position for the clockwise and counter clockwise UMBmark evaluation in reality, Unity and Gazebo.

for the ground truth results. For the odometry calculations, the computed, next position was always depended on the previous position and orientation. The calculations have been explained in more detail in Section 4.4.8. For matching this trajectory to the now modified ground truth trajectory, the starting position and orientation had just to be set to zero.

The absolute end position after each clockwise or counter clockwise run of the robot has been measured. This position was then compared to the position that the TurtleBot2 should have in regard to the odometry measurements. The difference in x- and y-position between those two end positions was calculated and plotted into a graph. The results of reality, Unity and Gazebo can be seen in Figure 24.

According to [47], the expected results are clusters of the calculated errors for each the clockwise and the counter clockwise direction. These clusters of errors are made up out of two different error sources, systematic and non-systematic errors. The systematic errors should approximately lie within the centre of gravity of that cluster. The non-systematic errors are the dispersion of the errors around that centre of gravity [47].

The dispersion of the measurements for the real robot was comparably large with an approximate difference of maximum 1.8 m. This illustrated a large amount of non-systematic errors in the measurements. Non-systematic errors vary between different measurements, unlike the systematic errors which stay constant. Non-systematic errors in this case could be different friction values due to dust on the wheels and on the ground or the current battery condition. It was observed during the measurements that the TurtleBot2 decreased its speed when the charge of the battery got too low. For measurements during which effects of low battery charge were observed, the data has been discarded. Nevertheless, the influence has to be taken into account. Smaller variation which did influence the movement of the robot but did not significantly decrease the speed of the robot to rise attention might have remained undiscovered during the conduction of the measurements.

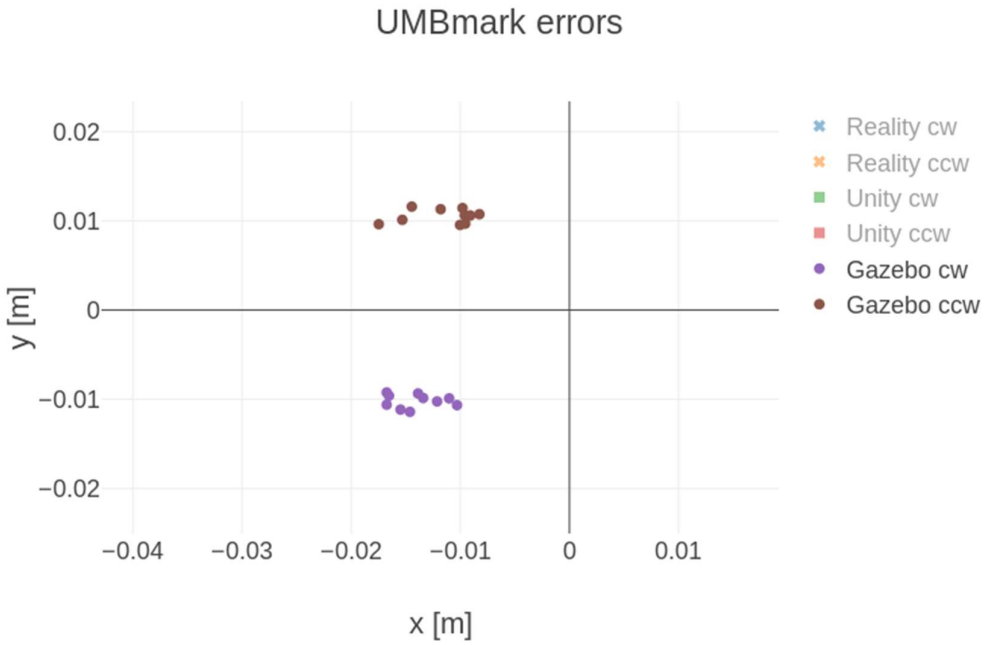


Figure 25. Enlarged extract of UMBmark results. Shows the errors of the Gazebo simulation in more detail.

The results for Unity were more regular and fit into the expected outcomes for an UMBmark. The errors for the clockwise and counter clockwise direction cluster in a group each. The errors show, in comparison to the errors of the results in reality, a higher precision. This indicated that the non-systematic errors of the measurements in Unity were smaller than in reality. The charge of the battery did not influence the Unity measurements, as the battery was not simulated here. The actuation of the wheels was done by a simulated motor, but the charge status was not taken into account. This supported the assumption that the high dispersion and therefore high non-systematic errors in reality could be related to the charge of the battery.

An even higher precision and accuracy can be seen in the results of Gazebo. The results are included in Figure 24, but hardly recognisable due to the huge deviation in precision and accuracy compared to Unity and reality. Out of that reason, an enlarged extract of the results in Figure 24 is presented in Figure 25. The clusters showed a dispersion of maximum 0.01 m. This was less than 1% of the dispersion that can be seen for the measurements with the real robot.

In addition to the error evaluations, the centre of gravity $[x_{c.g}, y_{c.g}]$ for each cluster was calculated. For computing the centre of gravity, the mean value in x- and y-direction of all errors within one cluster was calculated. The results were then plotted and can be seen in Figure 26. According to [47], the maximum odometry error was computed. For doing that, the error distance r was calculated by:

$$r = \sqrt{(x_{c.g})^2 + (y_{c.g})^2}.$$

This was done for the centre of gravity in clockwise and counter clockwise direction of each measuring environment. This resulted in two error distances for each reality, Unity and Gazebo. The maximum of the error distance r in clockwise and counter clockwise direction was defined as the maximum odometry error. The results are presented in

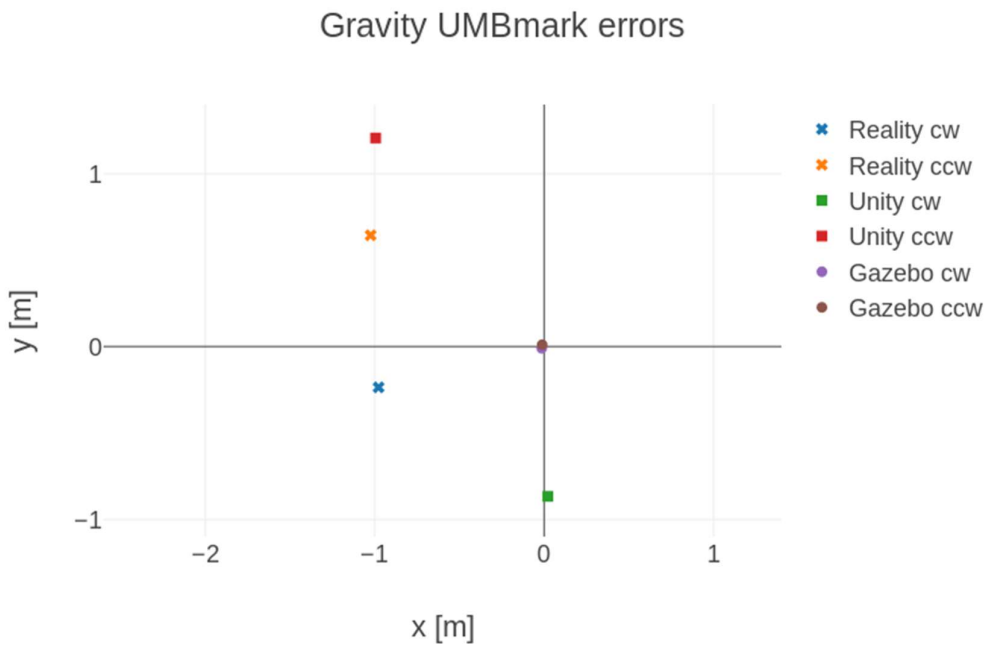


Figure 26. Gravity of the error clusters for the UMBmark test.

Table 2. These values illustrated the already observed differences in accuracy between the simulation in Gazebo and the results of the measurements in reality and in Unity. The maximum odometry error for Gazebo was significantly lower than the ones of the measurements in reality and Unity.

Table 2. Results from calculating the measure of dead-reckoning accuracy according to [47].

Environment	Maximum odometry error
Reality	1.06 m
Unity	1.56 m
Gazebo	0.02 m

Note that this does not imply that the simulation in Gazebo was more accurate than the simulation in Unity. The only statement that can be drawn out of these results was that the mismatch between ground-truth position and calculated position was significantly smaller and much more precise in Gazebo than in reality and Unity. The simulation in Unity seemed to more realistic, as the errors were in the same magnitude as the results from the measurements with the real robot.

5.9 SLAM case study

In the SLAM case study, it was investigated how well suited Unity and RosSharp were to conduct simulations with SLAM algorithms in cooperation with ROS nodes. The basic setup of this case study and why it was chosen was described in Section 4.4.9. When developing the experiments and setting up the environment in Unity and ROS, several problems appeared. One of them was a fundamental timing problem. The time in ROS is stated in two integers of time in seconds and time in nanoseconds since the beginning of the UNIX time at 1st January 1970. The time in RosSharp is also stated in two integers of time in seconds and time in nanoseconds, but the time here does not start at the beginning of the UNIX time. Instead, it is the real time since the start of the simulation, taken from the Unity parameter Time.realtimeSinceStartup. This was a problem, as the SLAM scripts in ROS used the timestamps of the messages in their calculations and interpret the messages from Unity as too old to be taken into account.

This mismatch could be bypassed by subscribing to the ROS clock in Unity and using the current ROS time for the timestamps of the Unity messages. It has to be taken into account that this was not an entirely correct way to solve this problem. The ROS clock as well as the Unity messages were published at a certain frequency. The subscription on Unity's side was event based and occurred whenever a new message of the wanted topic was published. Nevertheless, the delay between publishing the clock on ROS side, transferring the message via the rosbridge, subscribing to the clock on Unity's side and then using that timestamp for publishing the new messages was not taken into account. The main issue here was the need of different operating systems for Unity and ROS. If this would not be the case, there would not be the need to transfer the time over the network or even publish it. All nodes could just use the system time for their timestamps, as it would be possible to let all of them run on top of the same system.

After the experiments for this thesis and the SLAM case study were finished, a new feature regarding the time in Unity was committed to the RosSharp repository [56]. The

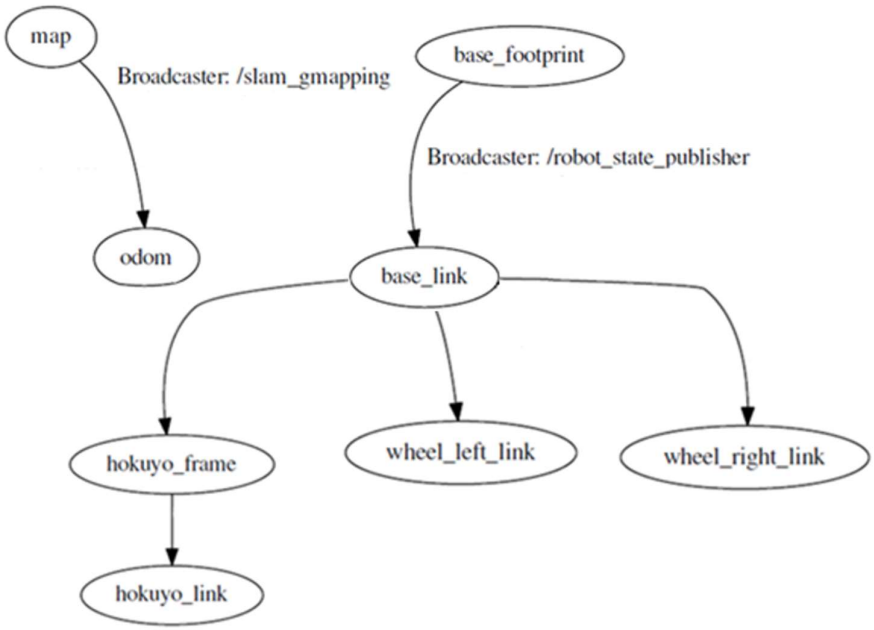


Figure 27. Simplified tf tree with robot_state_publisher and slam_gmapping.

new approach includes the possibility to switch between simulation time and system time for the timestamps. When using the simulation time, the previous implementation of RosSharp is used and the time is measured from the start of the simulation. When switching to the system time instead, the time since the start of the UNIX time is calculated for the current system time. It has to be noted that it is essential for this approach to have synced system times for both Windows and Linux.

Another problem that occurred was related to the tf trees. In a first attempt, the robot_state_publisher was used to calculate the tf of the robot from the base_footprint to all parts. This can be seen in Figure 27. The tf tree was simplified for this figure out of presentation reasons. The transforms for all plates, poles, the supporting wheels and

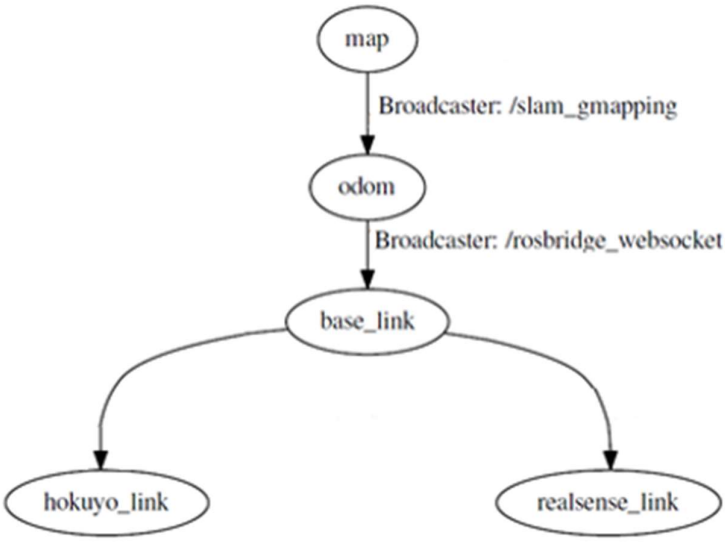


Figure 28. Tf tree with transforms calculated in Unity scripts and slam_gmapping.

the camera were excluded to improve the clarity of the figure. All subsequent parts to the base_footprint were broadcasted by the robot_state_publisher.

The resulting tf was to be used by the SLAM node. The SLAM node computed the transform between the map and the odometry. A missing part for this approach was the transform between the odometry and the base_footprint. This can be seen in the missing connection between odom and base_footprint in Figure 27. This was usually done by an extra node which calculated the odometry based on the data of the wheel encoders, the information of the robot's setup and even data from IMUs [57].

In a second attempt to compute all transforms, a script in Unity was written. This script used the velocity, size and position of the wheels in regard to the centre point of the robot in Unity to compute the odometry for the robot. The same calculations were used in other evaluations and have therefore already been described in Section 4.4.8. This odometry computation was published as the transform between odom and the base_link. The transforms from the base_link to the two sensors, hokuyo_link and realsense_link were taken from the Unity transforms directly and published as well. All these transforms that were broadcasted by the rosbridge_websocket, as this was the connection between Unity and ROS, were published in a tf message. This message was then, like in the previous attempt, subscribed by the SLAM node. The SLAM node used the tf and scan messages to update the map and believe of the pose, which resulted in an updated transform between the map and the odometry. This was then broadcasted by the SLAM node. The related tf tree of this approach can be seen in Figure 28.

For the final solution, the previous attempts were combined. The tf script in Unity was still used to now compute and publish only the connection between odom and the base_footprint. Everything in the tf tree below the base_footprint was published by the robot_state_publisher and therefore it was based on the URDF file. The same URDF file was used to create the robot model in Unity. In addition to that, the SLAM node used the transforms that have been published by Unity, respectively the rosbridge, and the robot_state_publisher to create the map and the transform between the map and

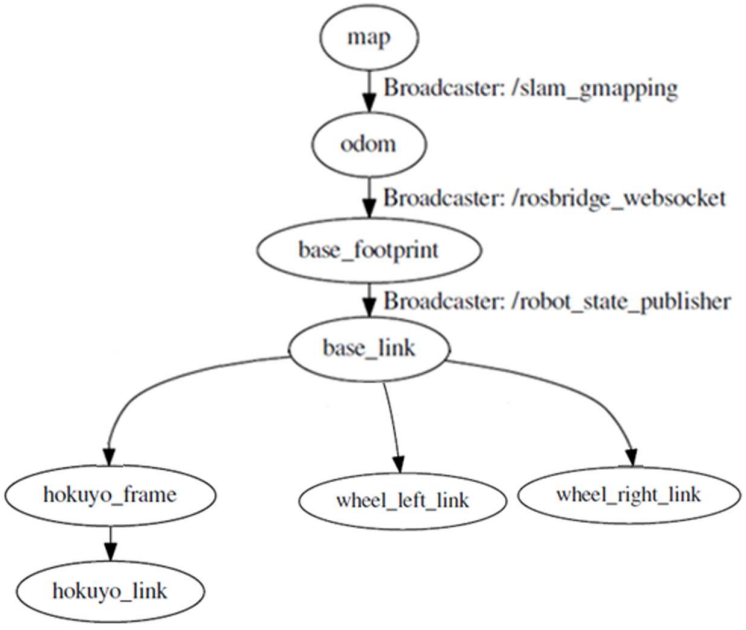


Figure 29. Simplified tf tree with transforms calculated in Unity scripts, robot_state_publisher and slam_gmapping

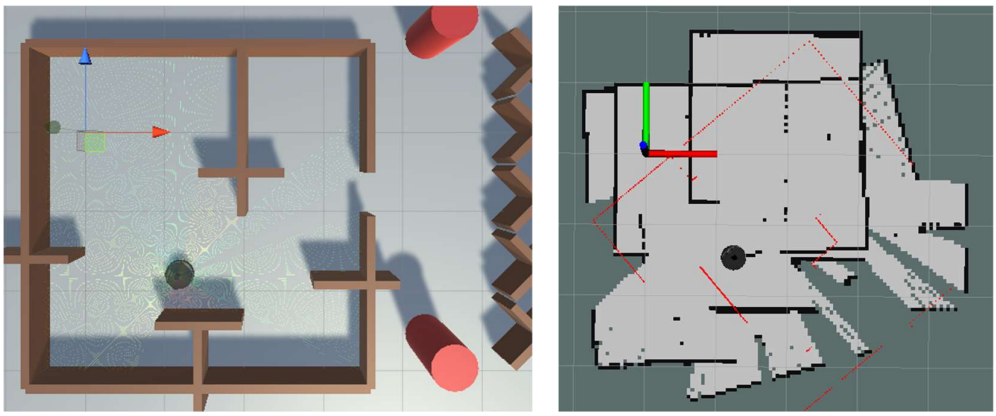


Figure 30. Simulation in Unity with laser scan visualisation (left) and resulting map of slam_gmapping node in RVIZ (right).

the odometry. A simplified version of the resulting tf tree can be seen in Figure 29. It does not include the transform between the base_link and the poles, plates, extra weight and the camera out of visualisation reasons.

This setup has been used for the SLAM case study. When publishing the laser scan messages from the RosSharp script LaserScanReader and running a simulation with a labyrinth, the map turned out to be not fitting. This behaviour can be seen in Figure 30. As soon as rotating the robot, the laser scans turned in a way that they did not fit in the previous map anymore. The turning of the laser scans can be seen in the right side of Figure 30. The laser scans are presented via the red lines and they do not fit with the map, which is shown in grey and black.

When taking a closer look to the LaserScanReader script of RosSharp, it was discovered that this incident was caused by the way the laser scans were measured. The rays, which represented the laser lines that were used for measuring, were defined as follows: the origin of the rays was dependent on the transform that they were attached to, in this case the laser link. Their direction on the other hand was dependent on the minimum and maximum angle of the laser, as well as the amount of laser samples and the increment between to angles. The current direction of the laser link in regard to the world coordinate system was not taken into account. That meant that no matter in what orientation the robot was at the moment, the laser scans would always start towards the same direction in the world coordinate frame. In reality, this was not the case. If the laser rotated, the direction of its samples would not be the same as before in regard to the world coordinate frame. To change the LaserScanReader script to make it match reality, the script was adjusted as presented in Figure 31.

```
1:          Rot = Quaternion.Euler(0,-angle_increment*i*180/π,0);
2:          Dir = Rot*transform.forward;
3:          Rays[i] = new Ray(transform.position,Dir);
```

Figure 31. Adjustments to LaserScanReader script to conduct dynamic laser scans.

Quaternion.Euler created a quaternion out of the Euler angles given in the subsequent brackets. In this case, the Euler angles were 0° around the x-axis and z-axis. The rotation around the y-axis was dependent on the angle increment between two samples and the counter i for the samples. In that way, the next sample was always one angle increment further away from the start than the current sample. The resulting quaternion Rot was then multiplied with transform.forward in the next line to get the direction of

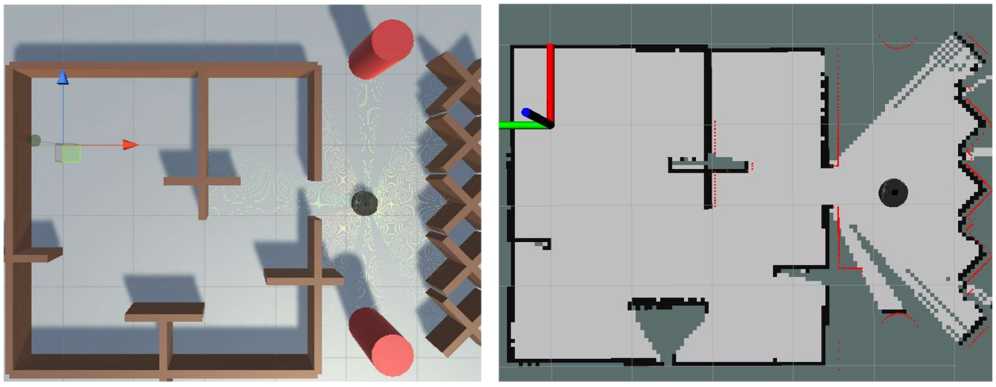


Figure 32. SLAM simulation in Unity with updated LaserScanReader script.

the laser line which was called Dir. Transform.forward represented the z-axis of the transform that the LaserScanReader was attached to. In the third line, a new ray was created. The inputs for the new ray were transform.position and the already calculated direction Dir. Transform.position was using the position of the game object that the LaserScanReader was attached to. In that way, the actual position of the game object and the direction of the laser line was used to sort and direct the laser scan measurements.

The LaserScanReader script in RosSharp has been changed accordingly and this incident was reported to the community [58]. When repeating the experiment with the changed script, the resulting map improved significantly, as can be seen in Figure 32. The laser scans rotated accordingly to the orientation of the robot and therefore fit in the map even when the robot was moved.

6 Conclusion

Simulation is getting increasingly popular in robotics research and brings several benefits for research and industry if it is of sufficient precision and accuracy. In this work, the simulation of mobile robots with Unity and ROS is examined and evaluated. Different, systematic test scenarios and evaluations parameters are determined and identified to investigate the performance of the simulation. Simulations of a mobile robot in Unity are compared to experiments with a mobile robot in reality and Gazebo. The underlying basic aim of this thesis work is to judge the performance of Unity as a simulator for mobile robots in comparison to reality and Gazebo. An associated question is if Unity, in combination with ROS, can be used for the simulation of mobile robots for research purposes.

The results of the qualitative benchmarks show that the simulation in Unity is able to reproduce the basic behaviour of the robot when it comes to friction and slopes. The behaviour of the μ-split benchmark could not be imitated with the real robot. This is probably due to too high friction values. Nevertheless, the robot is expected to show the rotation behaviour that it shows in the Unity and Gazebo simulations when the friction on one side becomes too low. It has to be taken into account that Unity does state to not necessarily have correct calculations when it comes to friction. The engine is tuned on realistic performance, not on physical correctness in this case [26].

The behaviour of the simulation in Unity during the ramp benchmark can also be explained and related to theory. The comparisons to the experiments with the real robot confirmed the basic behaviour of sideways movements and decreased speed. It has to be mentioned here that the while the sideways movement in Unity confirms with reality, the robot always turns towards the same direction in Unity. Why it shows that specific behaviour cannot be determined. A special behaviour of the robot, where it tipped over itself at higher slopes and fell down the ramp backwards, can be simulated neither in Unity, nor in Gazebo. This incident highlights the importance of a validated simulation model. The probable cause of this incident, the spring suspension of the actuated wheels, is not included in the used robot model. The behaviour of the simulated robot will obviously change if the robot model changes. The used model should be investigated carefully and validated before it is used in simulations. Only if a validated robot model is used and its assumptions and simplifications in regard to reality are known, the simulation results can be assessed in an appropriate way. The behaviour of the robot for the friction turn benchmark in Unity can be confirmed with reality.

The quantitative benchmarks show that the simulation in Unity is behaving more similar to the reality than Gazebo in this specific case. It has to be emphasised here that this does not necessarily retrace to the overall behaviour of Unity and Gazebo as simulator for mobile robots, as the performance of both simulators is highly dependent on the input parameters, settings and used models. Even though Gazebo shows lower errors and more idealistic behaviour, this correctness cannot be confirmed with reality. Parts of that idealistic behaviour can be retraced to the not existing simulation of a motor controller for the wheels. In Gazebo, the target velocity is directly set as actual velocity of the wheels. In that way, for example the theoretical end position in the basic drive benchmark can be reached in Gazebo. This is specifically mentionable, as the motor controller is not included in the widely used Gazebo simulation setup, but could

be easily extended with an additional controller script. In that way, this aspect of the idealistic behaviour would vanish and the simulation would be closer to reality and more comparable to the Unity results.

The simulation in Unity does not show such idealistic behaviour. A motor is simulated in the so-called hinge joints in Unity. The results include some unexpected behaviour, like the high curvature for the basic drive scenario at $0.3\ ^{m}/_{s}$ in Unity. It cannot be judged where this behaviour comes from. Even though the trend in those measurements is consistent for all measurement runs, the simulation is not deterministic.

The results of the UMBmark scenario show a realistic behaviour for the Unity measurements. They do not directly confirm with the results from reality, but are closer to the results of reality than the rather idealistic results from Gazebo in this specific case. The results from Unity show a combination of systematic and non-systematic errors, as expected according to the creators of the UMBmark [47]. The high dispersion of the results from the experiments with the real robot can be concluded to a rather huge amount of non-systematic errors which could be related to the battery level of the TurtleBot2.

The overall behaviour of the Unity simulation in all benchmarks is comparably good. The results in this specific case are similar to those with the real robot in most cases and the simulation is significantly less idealistic than the Gazebo measurements. The results of Unity show non-deterministic behaviour and it cannot be rated where this behaviour comes from. Sources for non-deterministic errors could be within ROS, the RosSharp connection and Unity. Even though the results were rather close to the results of reality, they don't justify the usage of simulation results of Unity for validation of the hardware configuration of a mobile robot. Small changes in the robot model and the simulation settings can have a huge impact on the results of a simulation and therefore it cannot be judged if the simulation results confirm with reality.

Instead, the simulation software can be combined with simulated controllers or algorithms. In that way, small deviations in the trajectory would not tip the balance for the results of the simulation. An example of the usage of Unity as a simulator for mobile robots and algorithms is shown in the SLAM case study. The scans of a simulated laser scanner can be used in combination with other data to create a map. If environments can be simulated and the resulting data be used for algorithms, this might especially come in handy when it comes to deep learning algorithms. Those algorithms are taught by the input of data and the simulation could be the main source for this needed data.

6.1 Future Work and Research

One critical aspect of simulation with Unity and ROS was the bridge that was needed between the two underlying operating systems. Problems that occurred here were for example related to timing aspects, as discussed in Section 5.9. The next version of ROS, ROS 2 is at the time of the completion of this thesis under heavy development [59]. Apart from many other innovations and changes in structure, ROS 2 is planned to be fully available on Windows. If ROS 2 were running on Windows and connected with Unity, the bridge between Windows and Linux would become redundant. This would most probably lead to better results and less timing- and other transferring issues. Future work regarding the simulation of mobile robots with Unity and ROS could therefore focus on the usage of ROS 2 with Unity.

6.2 Critical Discussion

During the work for this thesis, several critical parameters and aspects were discovered. Firstly, the parameter settings of Unity had a massive influence on the performance of the engine. In addition to that, the same parameter setting could cause different qualities in the outputs for different setup. That said, the same parameter setting could give very realistic results in one setup, but very unrealistic results in another setup.

This also reflects to the comparison between Unity and Gazebo. Both simulators and both underlying physics engines, PhysX for Unity and ODE for Gazebo, have a great amount of parameters and approaches that make a general comparison very difficult, if not impossible. The behaviour of the simulation and the results can be entirely different with just small changes of the input parameters. That said, the comparison between Unity and Gazebo is a small snippet with the default parameters and popular models used in the community rather than a thorough comparison.

6.3 Generalization of the result

When using a new robot, robot model or simulator, experiments like in this thesis are important for the assessment of the simulations' performance. Only if the validity and limitations of the simulation are known, the results can be used for evaluation or further processing. Of course, there does not always exist the possibility to do comparisons between the simulation of robots and the corresponding robots in reality, like done in this thesis with the TurtleBot2. Even if this is not possible, preliminary test can be done with similar robots or the results can be compared to theory or expected behaviour, like in the basic benchmarks and qualitative benchmarks of this thesis work. With the results, mistakes or simplifications in the simulator and the used model could be discovered.

Generally, the testing of simulators and used models is of significant importance for the usage of those parts, both in research and in industry. Therefore, the results and approaches of this thesis are for importance when it comes to simulation of mobile robots.

7 References

[1] International Organization for Standardization, *ISO 8373:2012 Robots and Robotic Devices - Vocabulary,* ISO, 2012.

[2] "Unity," Unity Technologies, [Online]. Available: https://unity.com/. [Accessed 01 06 2019].

[3] NVIDIA, "NVIDIA PhysX SDK 3.3.4 Documentation," 03 05 2016. [Online]. Available: https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/3.3.4/Index.html. [Accessed 25 02 2019].

[4] "Unity Machine Learning," Unity Technologies, 2019. [Online]. Available: https://unity3d.com/machine-learning. [Accessed 17 05 2019].

[5] C. Bartneck, M. Soucy, K. Fleuret and E. B. Sandoval, "The Robot Engine - Making the Unity 3D Game Engine Work for HRI," 2015.

[6] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar and D. Lange, "Unity: A General Platform for Intelligent Agents," 2018.

[7] M. Bloesch, Czarnowski, Jan, R. Clark, S. Leutenegger and A. J. Davison, "CodeSLAM - Learning a Compact, Optimisable Representation for Dense Visual SLAM," Imperial College London, 2018.

[8] "Gazebo," Open Source Robotics Foundation, 2014. [Online]. Available: http://gazebosim.org/. [Accessed 01 06 2019].

[9] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid and J. J. Leonard, "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age," *IEEE Transactions on Robotics,* vol. 32, no. 6, pp. 1309-1331, 2016.

[10] "ROS.org," Open Source Robotics Foundation, 08 08 2018. [Online]. Available: http://wiki.ros.org/ROS. [Accessed 25 04 2019].

[11] T. Foote, "ros.org," 07 2018. [Online]. Available: http://download.ros.org/downloads/metrics/metrics-report-2018-07.pdf. [Accessed 25 04 2019].

[12] W. Honig, C. Milanes, L. Scaria, T. Phan, M. Bolas and N. Ayanian, "Mixed Reality for Robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hamburg, 2015.

[13] A. Hussein, F. Garcia and C. Olaverri-Monreal, "ROS and Unity Based Framework for Intelligent Vehicles Control and Simulation," in *IEEE International Conference on Vehicular Electronics and Safety*, Madrid, 2018.

[14] "Rosbridge_suite," Open Source Robotics Foundation, 23 10 2017. [Online]. Available: http://wiki.ros.org/rosbridge_suite. [Accessed 06 05 2019].

[15] Y. Mizuchi and T. Inamura, "Cloud-based Multimodal Human-Robot Interaction Simulator Utilizing ROS and Unity Frameworks," in *IEEE International Symposium on System Integration SICE*, Taipei, 2017.

[16] R. Codd-Downey, P. M. Frooshani, A. Speers, H. Wang and M. Jenkin, "From ROS to Unity: Leveraging Robot and Virtual Environment Middleware for Immersive Teleoperation," in *IEEE International Conference on Information and Automation*, Hailar, 2014.

[17] M. Quigley, B. Gerkey, K. Conley, J. Faust and T. Foote, "ROS: an Open-Source Robot Operating System," in *ICRA Workshop on Open Source Software*, Kobe, 2009.

[18] "JointState Message," Open Source Robotics Foundation, 09 11 2018. [Online]. Available: docs.ros.org/api/sensor_msgs/html/msg/JointState.html. [Accessed 03 05 2019].

[19] "PoseStamped Message," Open Source Robotics Foundation, 09 11 2018. [Online]. Available: docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html. [Accessed 03 05 2019].

[20] "Twist Message," Open Source Robotics Foundation, 09 11 2018. [Online]. Available: docs.ros.org/api/geometry_msgs/html/msg/Twist.html. [Accessed 03 05 2019].

[21] "TransformStamped Message," Open Source Robotics Foundation, 09 11 2018. [Online]. Available: docs.ros.org/api/geometry_msgs/html/msg/TransformStamped.html. [Accessed 03 05 2019].

[22] T. Foote, "Tf2," 05 03 2019. [Online]. Available: http://wiki.ros.org/tf2. [Accessed 09 05 2019].

[23] R. Schollmeier, "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications," in *Proceedings First International Conference on Peer-to-Peer Computing*, Linkoeping, 2001.

[24] "ROS Installation Options," Open Source Robotics Foundation, 07 06 2018. [Online]. Available: http://wiki.ros.org/ROS/Installation. [Accessed 06 06 2019].

[25] W. A. Mattingly, D.-J. Chang, R. Paris, N. Smith, J. Blevins and M. Ouyang, "Robot Design Using Unity for Computer Games and Robotic Simulations," in *International Conference on Computer Games*, 2012.

[26] "Unity Manual (2018.2)," Unity Technologies, [Online]. Available: https://docs.unity3d.com/2018.2/Documentation/Manual/index.html. [Accessed 15 03 2019].

[27] "MuJoCo Plugin and Unity Integration," MuJoCo, [Online]. Available: http://www.mujoco.org/book/unity.html. [Accessed 30 04 2019].

[28] I. Deane, "Bullet Physics For Unity," Unity Asset Store, 01 02 2017. [Online]. Available: https://assetstore.unity.com/packages/tools/physics/bullet-physics-for-unity-62991. [Accessed 30 04 2019].

[29] N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, 2004.

[30] T. Erez, Y. Tassa and E. Todorov, "Simulation Tools for Model-Based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX," in *IEEE International Conference on Robotics and Automation*, Seattle, 2015.

[31] S. Peters and J. Hus, "Comparison of Rigid Body Dynamic Simulators for Robotic Simulation in Gazebo," in *ROSCon*, Chicago, 2014.

[32] S.-J. Chung and N. Pollard, "Predictable Behavior During Contact Simulation: a Comparison of Selected Physics Engines," *Computer Animation and Virtual Worlds,* vol. 27, no. 3-4, pp. 262-270, 2016.

[33] M. A. Sherman, A. Seth and S. L. Delp, "Simbody: Multibody Dynamics for Biomedical Research," *Elsevier,* vol. 2, no. IUTAM Symposium on Human Body Dynamics, pp. 241-261, 2011.

[34] A. Boeing and T. Bräunl, "Evaluation of Real-Time Physics Simulation Systems," 2007.

[35] S. Ivaldi, J. Peters, V. Padois and F. Nori, "Tools for Simulating Humanoid Robot Dynamics: a Survey Based on User Feedback," in *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, Madrid, 2014.

[36] E. Coumans, "Github," 2015. [Online]. Available: https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf. [Accessed 25 02 2019].

[37] E. Drumwright and D. A. Shell, "An Evaluation of Methods for Modeling Contact in Multibody Simulation," in *IEEE International Conference on Robotics and Automation*, Shanghai, 2011.

[38] A. Roennau, F. Sutter, G. Heppner, J. Oberlaender and R. Dillmann, "Evaluation of Physics Engines for Robotic Simulations With a Special Focus on the Dynamics of Walking Robots," in *International Conference on Advanced Robotics (ICAR)*, Montevideo, 2013.

[39] "Preparing Assets for Unity," Unity Technologies, 05 02 2019. [Online]. Available: https://docs.unity3d.com/2018.2/Documentation/Manual/BestPracticeMakingBelievableVisuals1.html. [Accessed 21 05 2019].

[40] A. Yakovlev, "Unity Blog; High-Performance Physics in Unity 5," Unity, 08 07 2014. [Online]. Available: https://blogs.unity3d.com/2014/07/08/high-performance-physics-in-unity-5/. [Accessed 08 04 2019].

[41] "ROS#," Siemens, 2017. [Online]. Available: https://github.com/siemens/ros-sharp. [Accessed 07 05 2019].

[42] "Index of /kinetic/api," Open Source Robotics Foundation, [Online]. Available: http://docs.ros.org/kinetic/api/. [Accessed 26 03 2019].

[43] T. Foote and M. Purvis, "Standard Unity of Measure and Coordinate Conventions," 07 10 2010. [Online]. Available: https://www.ros.org/reps/rep-0103.html. [Accessed 30 04 2019].

[44] M. Bischoff, "TransformExtensions.cs," 2018. [Online]. Available: https://github.com/siemens/ros-sharp/blob/master/Unity3D/Assets/RosSharp/Scripts/Extensions/TransformExtensions.cs. [Accessed 15 05 2019].

[45] "TurtleBot Description," Unknown, 12 12 2017. [Online]. Available: https://github.com/turtlebot/turtlebot/tree/kinetic/turtlebot_description. [Accessed 24 05 2019].

[46] Unknown, "Kobuki_Gazebo.urdf.xacro," 01 04 2017. [Online]. Available: https://github.com/yujinrobot/kobuki/blob/devel/kobuki_description/urdf/kobuki_gazebo.urdf.xacro. [Accessed 17 05 2019].

[47] J. Borenstein and L. Feng, "UMBMark- a Method for Measuring, Comparing, and Correcting Dead-Reckoning Errors in Mobile Robots," 1994.

[48] B. Siciliano, L. Sciavicco, L. Villani and G. Oriolo, Robotics: Modelling, Planning and Control, London: Springer-Verlag London Limited, 2009.

[49] S. Thurn, W. Burgard and D. Fox, Probabilistic Robotics, Cambridge: The MIT Press, 2006.

[50] J. Craighead, "SICK Laser Scanner," 10 01 2012. [Online]. Available: http://wiki.unity3d.com/index.php/SICK_Laser_Scanner. [Accessed 07 05 2019].

[51] Unknown, "IMU Inertial Measurement Unit," 10 01 2012. [Online]. Available: http://wiki.unity3d.com/index.php/IMU_Inertial_Measurement_Unit. [Accessed 07 05 2019].

[52] B. Gerkey and V. Rabaud, "Gmapping," 04 02 2019. [Online]. Available: http://wiki.ros.org/gmapping. [Accessed 10 05 2019].

[53] "SDFormat," Open Source Robotics Foundation, 2019. [Online]. Available: http://sdformat.org/spec. [Accessed 14 05 2019].

[54] "Gazebo Friction," Open Source Robotics Foundation, 2014. [Online]. Available: http://gazebosim.org/tutorials?tut=friction. [Accessed 15 05 2019].

[55] D. Hewlett and A. Rebguns, "Gazebo_ros_diff_drive.cpp," 2010. [Online]. Available: https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_diff_drive.cpp. [Accessed 14 05 2019].

[56] Unknown, "Adding Timestamp Switching RosSharp," 15 05 2019. [Online]. Available: https://github.com/siemens/ros-sharp/pull/200. [Accessed 20 05 2019].

[57] W. Meeussen, "Coordinate Frames for Mobile Platforms," 28 10 2010. [Online]. Available: http://www.ros.org/reps/rep-0105.html. [Accessed 08 05 2019].

[58] "Static Laser Orientation in LaserScanReader," 08 05 2019. [Online]. Available: https://github.com/siemens/ros-sharp/issues/199. [Accessed 14 05 2019].

[59] Unknown, "ROS 2," 05 04 2019. [Online]. Available: https://index.ros.org/doc/ros2/. [Accessed 16 05 2019].

[60] "NVIDIA PhysX SDK," Nvidia, 2019. [Online]. Available: https://github.com/NVIDIAGameWorks/PhysX. [Accessed 02 06 2019].

[61] R. L. Smith, "ODE," 2007. [Online]. Available: https://bitbucket.org/odedevs/ode/src/default/. [Accessed 02 06 2019].

[62] "Bullet Physics SDK," Unknown, [Online]. Available: https://github.com/bulletphysics/bullet3. [Accessed 02 06 2019].

A. Software and Hardware Setup

Table 3. Software and hardware used for Unity simulations.

Software/Hardware	Version
Processor	I9-7940X 3.10 GHz
Cores	14
RAM	32.0 GB
Operating system	Windows 10 Pro 1803
Unity	2018.2.17f1
PhysX	3.3.3
VM	Oracle VirtualBox 6.0.2
VM Operating system	Ubuntu 16.04.6 LTS
VM ROS	Kinetic
VM Cores	4
VM RAM	8192 MB

Table 4. Software and hardware used for Gazebo simulations.

Software/Hardware	Version
Processor	E3-1505M 2.80 GHz
Cores	8
RAM	32.0 GB
Operating system	Ubuntu 16.04.6 LTS
ROS	Kinetic
Gazebo	9.9.0
ODE	https://bitbucket.org/osrf/gazebo/src/default/deps/opende/

Table 5. Software and hardware used for the experiments in reality.

Software/Hardware	Version
Robot	TurtleBot 2e
Robot base	Kobuki
Motion Capture Software	Motive 2.0.2

Table 5. Parameter settings used for Unity simulations.

Parameter	Value
Gravity	0, -9.81, 0
Static Friction	1.4
Dynamic Friction	1.4
Bounciness	0
Friction Combine	Maximum
Bounce Combine	Average
Bounce Threshold	2
Sleep Threshold	0.005
Default Contact Offset	0.01
Default Solver Iterations	6
Default Solver Velocity Iterations	1
Queries Hit Backfaces	False
Queries Hit Triggers	True
Enable Adaptive Force	False
Contacts Generation	Persistent Contact Manifold
Auto Simulation	True
Auto Sync Transforms	True
Contact Pairs Mode	Default Contact Pairs
Broadphase Type	Sweep And Prune Broadphase
World Bounds Center	0, 0, 0
World Bounds Extent	250, 250, 250
World Subdivisions	8
Layer Collision Matrix	All true
Cloth Inter-Collision	False
Fixed Timestep	0.02
Maximum Allowed Timestep	1
Time Scale	1
Maximum Particle Timestep	0.03

B. Robot Model

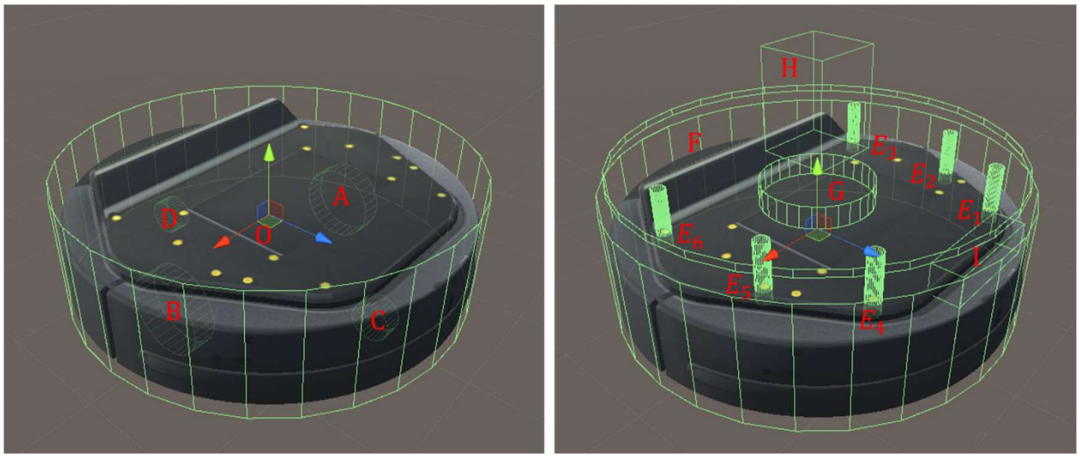


Figure 33. Setup of TurtleBot2 model with naming declarations of links.

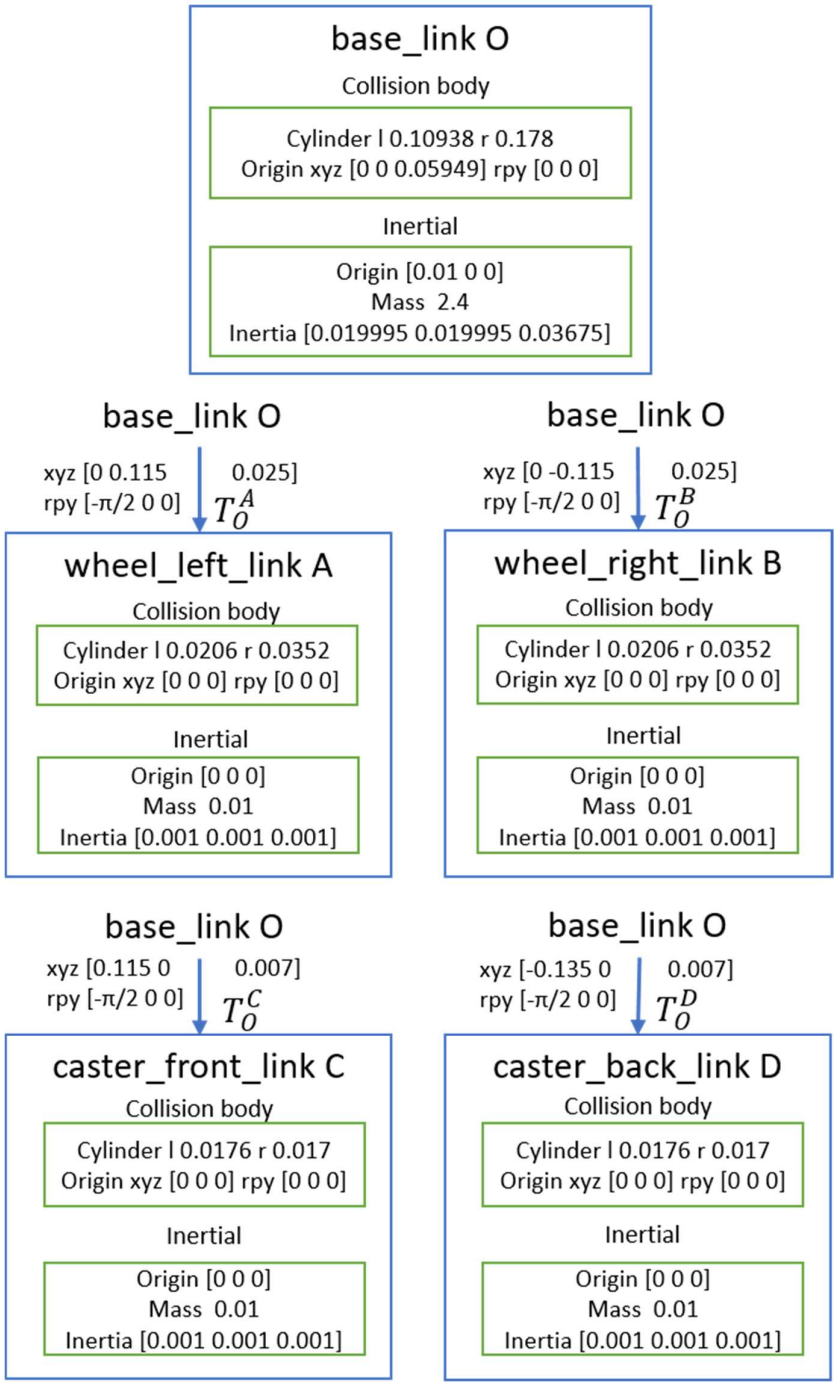


Figure 34. Dynamics of used URDF model. Extract 1.

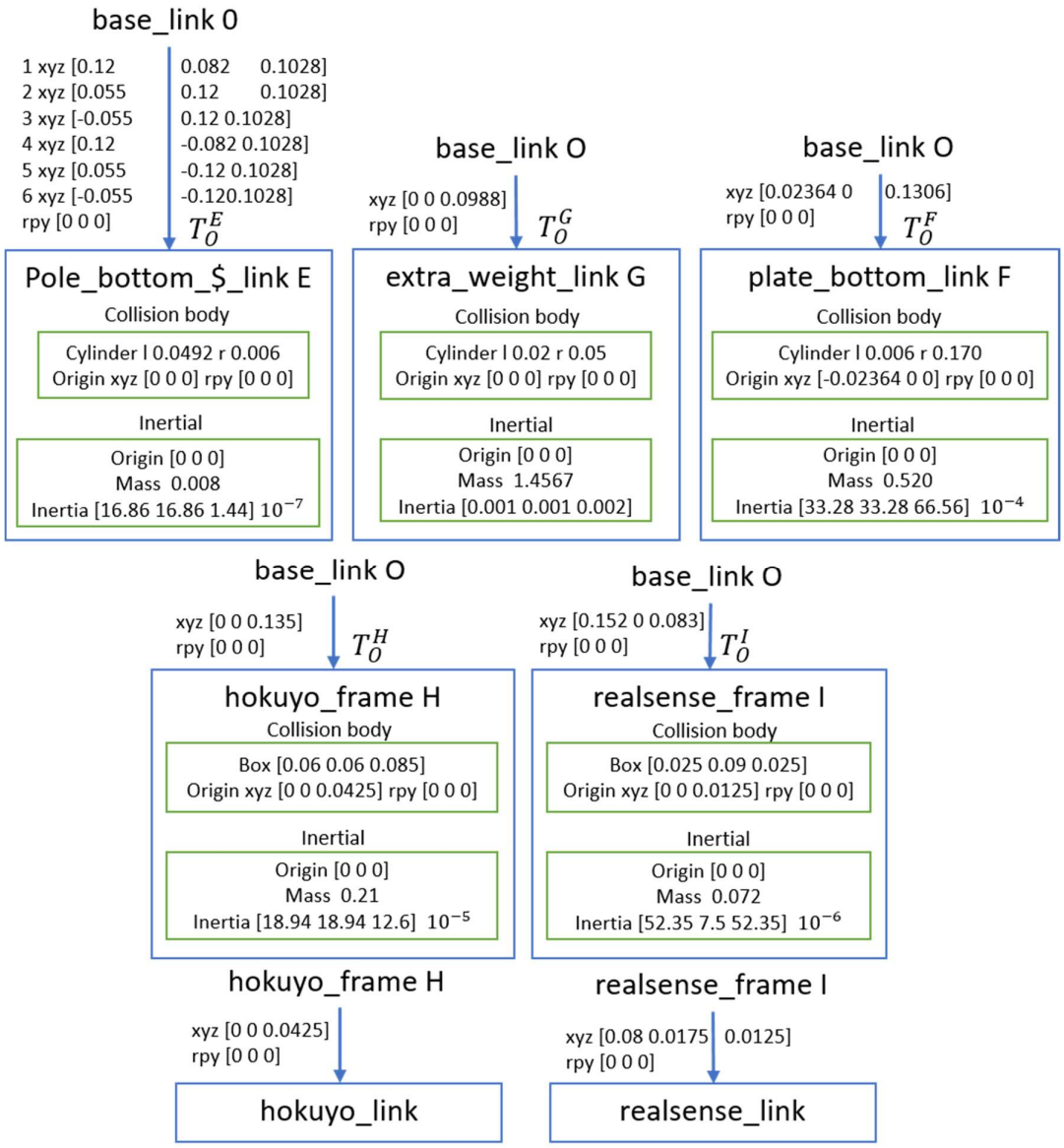


Figure 35. Dynamics of used URDF model. Extract 2.

C. C# Script for Wheel-Motor in Unity

```
using UnityEngine;

namespace RosSharp.RosBridgeClient
{
    0 references
    public class TwistSubscriberMotorDriver : Subscriber<Messages.Geometry.Twist>
    {
        public float w_radius = 0.0352F; // [m]
        public float w_distance = 0.230F; // [m]
        public HingeJoint wheelleft;
        public HingeJoint wheelright;

        private float vel_left; // [°/s]
        private float vel_right; // [°/s]
        private bool isMessagerecieved = false;
        private JointMotor jointMotorleft;
        private JointMotor jointMotorright;
        private Vector3 vel_lin; // [m/s]
        private Vector3 vel_ang; // [m/s]
        16 references
        protected override void Start()
        {
            base.Start();
            wheelleft.useMotor = true;
            wheelright.useMotor = true;
            jointMotorleft = wheelleft.motor;
            jointMotorright = wheelright.motor;
        }
        0 references
        private void Update()
        {
            if (isMessagerecieved)
                Process();
        }
        1 reference
        private void Process()
        {
            jointMotorleft.targetVelocity = vel_left;
            jointMotorright.targetVelocity = vel_right;
            wheelleft.motor = jointMotorleft;
            wheelright.motor = jointMotorright;
            isMessagerecieved = false;
        }
        2 references
        private static Vector3 ToVector3(Messages.Geometry.Vector3 geometryVector3)
        {
            return new Vector3(geometryVector3.x, geometryVector3.y, geometryVector3.z);
        }
        10 references
        protected override void ReceiveMessage(Messages.Geometry.Twist message)
        {
            vel_lin = ToVector3(message.linear).Ros2Unity();
            vel_ang = ToVector3(message.angular).Ros2Unity();
            vel_right = 180/Mathf.PI*((2 * vel_lin.z + vel_ang.y * w_distance) / (2 * w_radius));
            vel_left = 180/Mathf.PI*((2 * vel_lin.z - vel_ang.y * w_distance) / (2 * w_radius));
            isMessagerecieved = true;
        }
    }
}
```

Figure 36. C# script in Unity to calculate the resulting angular velocity of the wheels from the linear and angular target velocity commands.