Comparison of Robotic Simulation Environments

Bachelor's Final Project

Department of Mechanical Engineering
Control System Technology

July 10, 2022

E.J.L. Wolfs

1439537

Supervisor:
Dr. E. Torta

Eindhoven University of Technology / Technische Universiteit Eindhoven
Faculty of Mechanical Engineering / Faculteit Werktuigbouwkunde

Table of Contents

1 Introduction

In today's world, robots have become increasingly important, more complex, and applied in many different fields. Robots are also being given more responsibility. Especially for complex medical robot applications where reliability and consistency are key. The robot should perform exactly what it is expected to do. Due to this complexity, optimisations in terms of controller algorithms and software are key factors to make sure that the robotic system works as intended. Validating these systems on real hardware is expensive and can therefore be executed less often. To improve this optimisation process and validate the robot before implementing it on a real-life setup, robotic simulation environments are used. These environments allow modelling of the robot in a realistic real-world environment, reducing the time and costs of the design cycle [2]. According to a previous study [8], these environments make it easier to solve problems with algorithms due to their great controllability, which makes it possible to reduce the number of factors involved in a simulation, such as friction. Another advantage is the fact that simulation environments are more predictable compared to the real world. This ensures that the results of the experiments are always constant.

When validating a robot, it is important to choose a suitable simulation environment, since different environments offer different built-in features. Due to the increasing number of simulation environments and features, it is sometimes unclear which environment is best suitable for a specific robotic simulation task. There is a wide range of different robot types like mobile robots, humanoid robots, and manipulators. All these types require different simulation capabilities in terms of, for example, sensor implementation, collision modelling or environment simulation. Several studies have examined the differences between simulation environments. One of them is a study about the Analysis and Comparison of Robotics 3D Simulators [3]. This study compares the environments V-Rep, Unity and Gazebo by focusing mostly on the quality and usability of these environments. This research concluded that V-Rep has more integrated features, while Gazebo requires more plugins. This type of study gives already an indication of what environment might be more convenient to use for a specific simulation scenario.

The Robotics System Toolbox in Matlab provides tools for designing, simulating, and testing the robots in one integrated environment. To visualise and test these tools including complex integrated cameras and sensors, a simulation environment is needed. A 3D realistic multibody simulation environment like Simscape Multibody can be used for this [16]. This physical modelling tool is integrated into Simulink and automatically generates a 3D animation of the robotic setup. Making it convenient to directly test algorithms on the robot. MathWorks proves that it is also possible to use an external robotic simulation environment like Gazebo [12] together with Simulink, making use of co-simulation. Via co-simulation, Simulink and Gazebo are directly connected and can exchange data. This makes Simulink not limited to Simscape Multibody only. Based on this fact, the following question can be asked: "What are the main differences between simulating a robotic scenario in Simscape Multibody and Gazebo via co-simulation with Simulink?". This gives rise to the main objective of this study:

1. *Compare Simscape Multibody and Gazebo based on defined comparison criteria by making use of co-simulation with Simulink.*

In order to compare the simulation environments systematically, comparison criteria are needed. These criteria should be drawn up based on of the performance indicators to be evaluated. A suitable scenario must then be chosen where these criteria can be tested for. The same scenario can then be used for the simulation environments, to allow for a fair comparison. This gives rise to the following sub-objective of this study:

1a. Define comparison criteria and a scenario in order to compare the simulation environment based on utility, usability and performance.

One requirement of these environments is that they are able to co-simulate with Simulink. This means that Simulink should be able to send inputs (for example, position, velocity and torque) and receive outputs data from the robot (for example, position and sensor data) modelled in a specific simulation environment. To perform co-simulation, a connection between Simulink and Gazebo needs to be made. This leads to the following sub-objective of this study:

1b. Establish a connection between Gazebo and Simulink to perform a co-simulation with the defined scenario.

After comparing the two environments, the advantages of sensor simulation in Gazebo will be further elaborated based on a use case perspective. This use case is based on the implementation of a Model Predictive Control (MPC) algorithm for an Image-guided Therapy (IGT) robot. For this project, it is necessary to simulate synthetic sensor data to estimate the position of obstacles in a room, which can then be further used for obstacle avoidance. This gives rise to the second main study objective:

2. Investigate how to create a grid map of sensor simulation in Gazebo based on the requirements for the use case.

In Chapter 2, the comparison criteria are elaborated together with the definition of the scenario. The co-simulation that is used to simulate the scenario in Gazebo is further explained in Chapter 3. The comparison between Simscape Multibody and Gazebo is elaborated in Chapter 4 and is divided into different sections. Examples are demonstrated with the scenario and tests are done to find the differences between the environments. At the end of each section, a conclusion is made that reflects the main findings of the scenario. Finally, Chapter 5 describes the perception in Gazebo applied to the use case. Implementation of 2D and 3D grid maps are explained and tested with the IGT robot. The report ends with Chapter 6, which provides a conclusion and a recommendation.

2 Comparison Setup

This chapter describes the comparison criteria that serve as guidelines for comparing the simulation environments. Based on these criteria, a suitable scenario is chosen which is then used to compare the environments equally. The scenario description consists of an explanation of the chosen robot model including a visualisation. Furthermore, the main setup of the Simulink model used for the comparison is explained.

2.1 Comparison Criteria

To compare the simulation environments, several comparison criteria have been established. Because the differences between the environments can be analysed in many different ways, they are divided into different sections. This makes it clear which aspects are looked at when comparing. For this study the comparison criteria are subdivided into three main sections:

- **Utility**: The functionalities that the simulation environment offers in terms of simulation possibilities. For example, whether it is possible to include built-in sensor models or different scene objects in the simulation.
- **Usability**: How well users can execute/develop the functionalities that are possible with the environment. For example, how convenient it is to install or use the software and what knowledge is required to accomplish this.
- **Performance**: Examines the quality of the simulation in the environment. An example of this could be the efficiency and accuracy of the computations. Specifications such as CPU core usage, simulation time, and memory usage can be compared.

Based on these three sections, different criteria are chosen to be evaluated. The criteria that form the basis of the comparison can be seen in Table 2.1.

Table 2.1: Comparison criteria

Criteria	**Section**	**Description**
General Differences	Utility / Usability	The general characteristics and main features of the environments.
Simulation Workflow	Usability	The general workflow that is needed to perform a simulation within the environments, based on the defined scenario.
Sensors	Utility	The possibility to simulate virtual sensors including the type of sensors that can be simulated.
Joint Limits and Collision Modelling	Utility	Comparing the simulation behaviour of the scenario in terms of joint limits and collision modelling.
Simulation Time	Performance	The differences in simulation time between the environments, based on the scenario.

For this study, the two simulation environments Simscape Multibody and Gazebo will be compared by using co-simulation with Simulink. The defined comparison criteria form the basis of the comparison and will be further elaborated in the following sections.

2.2 Manipulator Scenario

Based on the comparison criteria described in Section 2.1 a scenario is chosen that is suitable to compare the environments. First, a general description of the scenario is given that forms the basis for the comparison which is equal for both environments:

A robotic manipulator is fixed at the base within an empty environment that includes gravity. The manipulator consists of four revolute joints and two prismatic joints which can move independently. The four revolute joints are connected to the arm of the manipulator and the two revolute joints are connected to the gripper. At the end of the manipulator, an end-effector is attached with a gripper that can be controlled. This gripper can be closed and opened completely. By making use of co-simulation between the simulation environment and Simulink, the reference trajectory and controllers are defined within Simulink. Meanwhile, the joint position data is received from the simulation environment.

2.2.1 Robot Representation and Simulink Model

The above-defined robot description is referred to as the OpenManipulator [26]. On the community page of MathWorks, a Simulink model is provided including the URDF (Unified Robot Description Format) files and STL (Standard Triangle Language) files of the OpenManipulator robot [17]. The Simulink model makes use of tools from the Robotic System Toolbox, Simsacape Multibody Multiphysics and Simcape Multibody Contact Forces library. In addition, the URDF of the robot model is implemented in a Simscape Multibody environment. This robot consists of four revolute joints which are attached to the links of the robot arm and 2 prismatic joints which are connected to the gripper links, see Figure 2.1. In this Figure the URDF of the robot is visualised in Matlab, showing the coordinate systems for the fixed (purple axis) and movable joints (RGB axis).

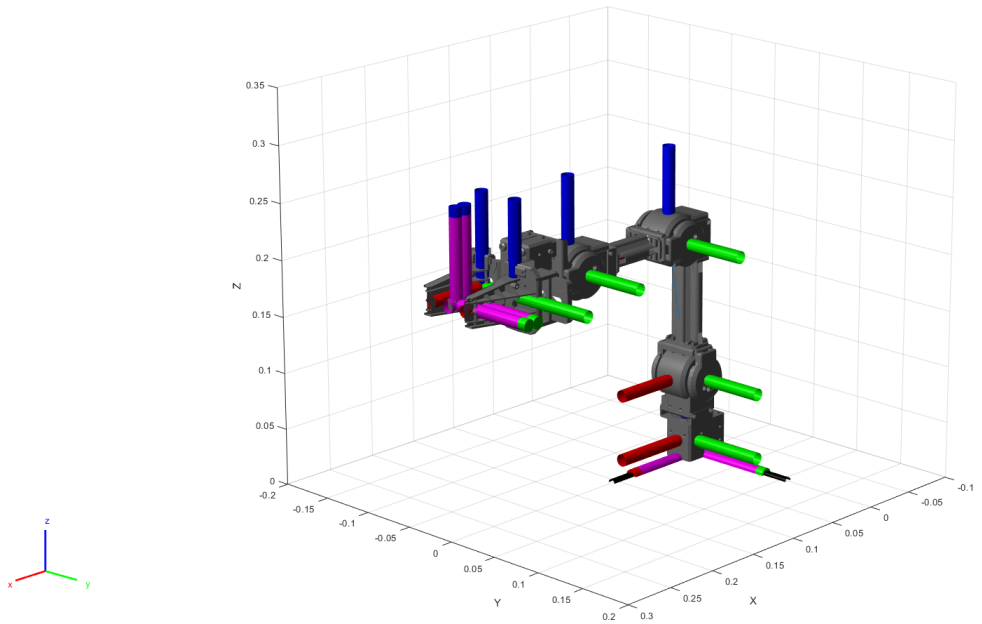


Figure 2.1: URDF of OpenManipulator

To perform an equal comparison, the Simulink models of the manipulator should be the same. In this model, the trajectory and controllers are defined and will therefore be equal for both simulation environments. Having the Simulink model, two separate models are made. One contains the plant in Simscape Multibody and the other consists of a co-simulation connection with Gazebo. The plant is referred to as the psychical model where the description of the robot is defined and the corresponding behaviour of the robot is simulated.

In Figure 2.2 a schematic representation of the comparison setup can be seen when simulating the OpenManpulator in Simscape Multibody. It can be seen that this environment is located within the Simulink environment (in Windows) and that no extra connection is needed between the Simulink model and the simulation environment.

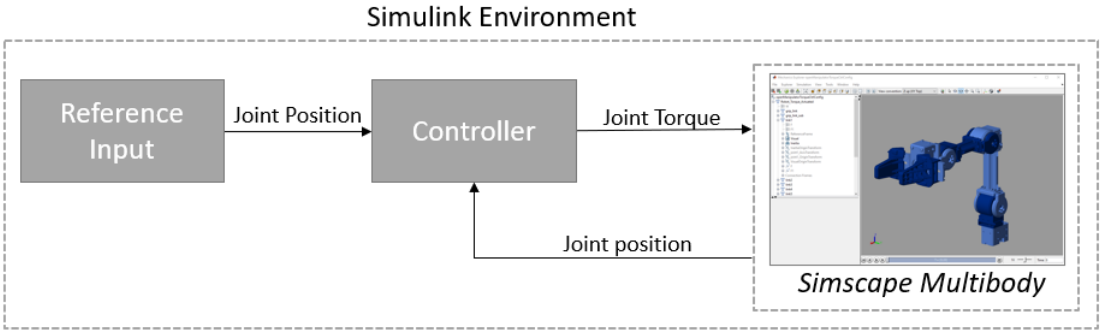


Figure 2.2: Controller scheme using Simscape Multibody environment

The schematic representation of the OpenManipulator in Gazebo can be seen in Figure 2.3. Gazebo is installed on a virtual machine running on the Linux operating system (Ubuntu). A connection between the virtual machine and the Simulink environment is needed to perform co-simulation.

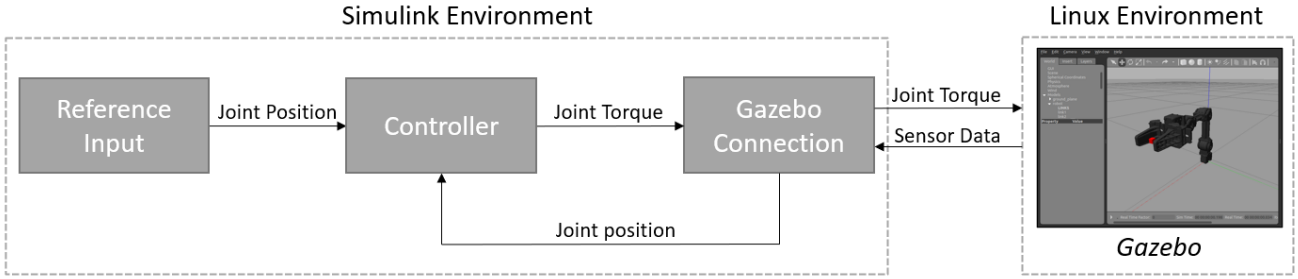


Figure 2.3: Controller scheme using Gazebo environment

Different Simulink models are used that are adjusted for each comparison, for example, to include sensor information. Some models use the reference position directly to move the robot, others contain a controller that calculates the torque for a given position as input. In Appendix A.4, the Simulink model is shown which forms the basis for the comparison. The plant model can be changed between the Simscape Multibody and the Gazebo environment. Four sine-waves are used as input signals for the four revolute joints and a step signal is used to move the gripper. These signals first go to the controller where the corresponding torques are calculated. The Simulink model in the documents from the OpenManipulator [17], provides a simple controller consisting of feedback PD controllers and a feed-forward controller. The feedforward controller uses the Feedforward Controller Block from the Robotic System Toolbox which calculates automatically the torques needed for each joint to track the reference signal. This block uses the URDF model of the robot, where information about the inertia and the masses of the links are defined. For the feedback controller, six PD controllers are used to control the four arms separately and the two gripper links. The feedback data comes from one of the simulation environments. Finally, visualisation plots are used to visualise the reference trajectory and the behaviour of the robot. These models together with the model in Figure A.5 can be found in the repository [43]. Chapter 3 further explains how the connection between Gazebo and Simulink was established.

3 Co-Simulation Simulink and Gazebo

In this chapter, it is explained how the OpenManipulator robot is implemented in a Gazebo environment and how it is connected to Simulink for co-simulation. Background information about the used plugin is also included. The general workflow for this setup is based on the Mathworks support documentation [20].

3.1 Virtual Machine with ROS Environment

In order to use Gazebo together with a ROS environment (see Appendix A.1 for a short explanation), a virtual machine "VMware workstation" is needed to run the Ubuntu operating system. Even though it is possible to run this software on Windows, it is preferred to use a Linux environment. For this study, a pre-defined virtual machine is used provided by MathWorks [21]. This virtual machine includes ROS 2 Dashing, ROS Melodic and Gazebo, see Appendix A.2 for more information. In addition, Geany (integrated development environment) and Terminator (a more advanced Linux terminal) are installed since they are convenient when using a ROS environment. The machine then provides enough tools to be used for the simulation of the manipulator scenario. After downloading these programmes, other plugins present on the machine were updated. The ROS build system is changed from catkin_make to catkin build. This ROS build system gives an isolated environment, making the build configuration more robust for changes in the catkin workspace. The complete virtual machine can be found in the repository [48].

3.2 OpenManipulator into Gazebo

To compare the simulation environments, a suitable example robot model was chosen named the "OpenManipulator" as explained in Chapter 2. This robot consists of a robotic platform including OpenSoftware, OpenHardware [10], and an e-manual [26] that explains the usages of the corresponding software. All robot files needed for the robot description and ROS-controller can be found on GitHub [7]. The OpenManipulator files from GitHub were downloaded to the virtual machine and a new ROS package was created to set up the robot environment. Inside this package, the launch file and the world file are located. A launch file is written in XML and can start programs like Gazebo together with multiple ROS nodes at once. The world file consists of an SDF format (Simulation Description Format), an XML code that describes not only the URDF of the robot but also contains all elements present in the Gazebo simulation world, like obstacles and sensors. This world file can then be opened with the launch file using the `roslaunch` command. After the environment with the OpenManipulator was configured in Gazebo, the connection between Simulink and Gazebo was established.

3.3 Connection between Simulink and Gazebo

To perform co-simulation between Simulink and Gazebo, a connection needs to be made. For this connection, two main elements are important: the Robotic System Toolbox in Matlab (including Simulink) and a Gazebo plugin. The Robotic System Toolbox provides robotic algorithms in Matlab and Simulink for the co-simulation framework with Gazebo. The Gazebo plugin is provided by MathWorks that consists of scripts which make sure that there is a data transfer between Simulink and the Gazebo environment. It also provides eight different world examples which can be loaded into Gazebo. The Gazebo plugin was downloaded and installed on the virtual machine via the MathWorks instructions [20]. Consequently, the plugin was added to the before-created world file of the OpenManipulator, see Appendix C.1 for the code.

3.3.1 Gazebo Co-simulation Plugin

Below in Figure 3.1, it can be seen how the data transfer of the co-simulation works in general. The output data coming from the Simulink environment, which is located on the host computer, is first sent to the Gazebo co-simulation plugin [19]. The plugin then passes the data to the Gazebo simulator. This plugin together with Gazebo is located in the Linux environment. During the simulation, the data is sent back to the Gazebo plugin which then sends it back to Simulink located on the host computer. The plugin ensures that Simulink and Gazebo are synchronised and run at the same pace.

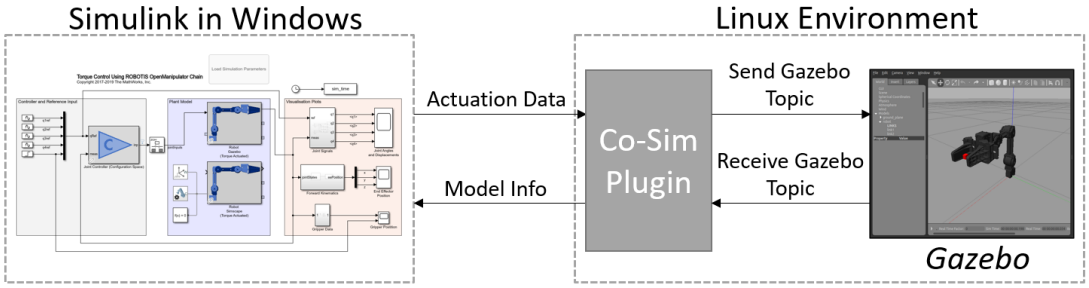


Figure 3.1: Communication between Simulink and Gazebo [19]

An advantage of this plugin is that it does not require the creation of a ROS network. Installing the plugin and adding it to the world file is enough to control the robot via Simulink. No additional ROS nodes with topics are needed because the plugin automatically sends the data to the Gazebo topics. One limitation of this plugin is that it does not support code generation. This means that it is not possible, for example, to create a ROS node directly from the existing Simulink model used for the co-simulation.

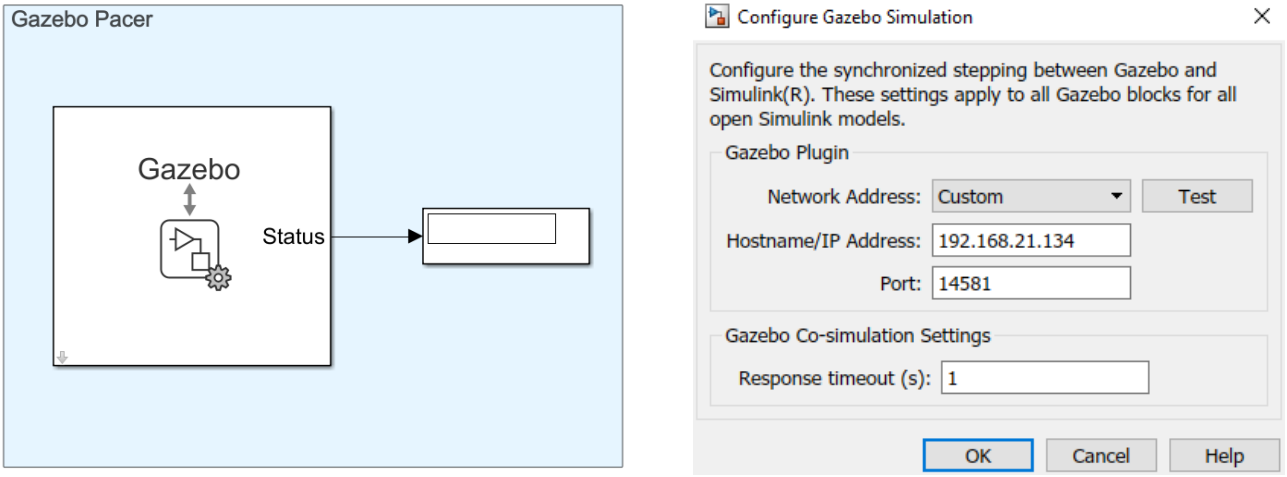
Instead of using the Gazebo plugin, it is also possible to connect Simulink with a ROS network via the ROS Toolbox [38]. Simulink can then publish or receive messages by subscribing to a ROS topic from a ROS network that can be connected to Gazebo. With this method, the Simulink and Gazebo are not synchronised. This means that Simulink can publish ROS topics at different rates compared to the updating rate of Gazebo. Via this configuration, it is possible to generate C++ codes for stand-alone ROS nodes that can be directly uploaded to the ROS environment.

3.3.2 Connection Setup

After the plugin has been successfully installed and configured for the scenario, the connection to Matlab is made. To do this, the IP of the virtual machine is required. This IP is found using the `ifconfig` command in the terminator of the Ubuntu environment. The Matlab co-simulation function `Gzinit` is used to initialise the connection between the local host and the Gazebo plugin. Accordingly, the port number "14581" is specified in the XML code of the world file and is then used for the Matlab function to make a connection with Gazebo, as can be seen in the following code:

```
ipGazebo = '192.168.21.134';    % Virtual Machine IP
gzinit(ipGazebo,14581);         % Initialise connection
```

To establish the connection with Simulink, the Gazebo Pacer block is used, see Figure 3.2a. This block is from the Robotic System Toolbox library which can be found in the Gazebo co-simulation section of the Simulink library. In this block the IP of the virtual machine and the port number are set, Figure 3.2b. Having the Gazebo simulation environment running the connection can be tested here. This connection block is used for all Simulink files, only the host name needs to be changed when using a Simulink file from the repository [43].



(a) Gazebo pacer block *(b) Configuration of connection*

Figure 3.2: Configuration of Simulink connection

3.3.3 Complete Co-Simulation Connection

The complete model that connects the OpenManipulator in Gazebo with Simulink is shown in Figure 3.3. This system consists of two main parts. The left part sends data to Gazebo and consists of six sending blocks each connected to a different movable joint of the OpenManipulator. The right part receives data from Gazebo and consists of six receive blocks and is also connected to each joint individually. The data of the joints received from Gazebo is subsequently filtered using a bus selector block giving only the position as output. Finally, a rate transition block is used to account for the possible differences in data rates between the Gazebo and the Simulink model which can otherwise lead to data integrity. A detailed description of the connection setup can be found in Appendix A.3.

This connection setup forms the basis for connecting Simulink with Gazebo, and it can be modified in order to use also velocity or torque as an input. In the repository [40], different examples are located that show how different parts of the OpenManipulator can be controlled using position, velocity and torque as an input signal.

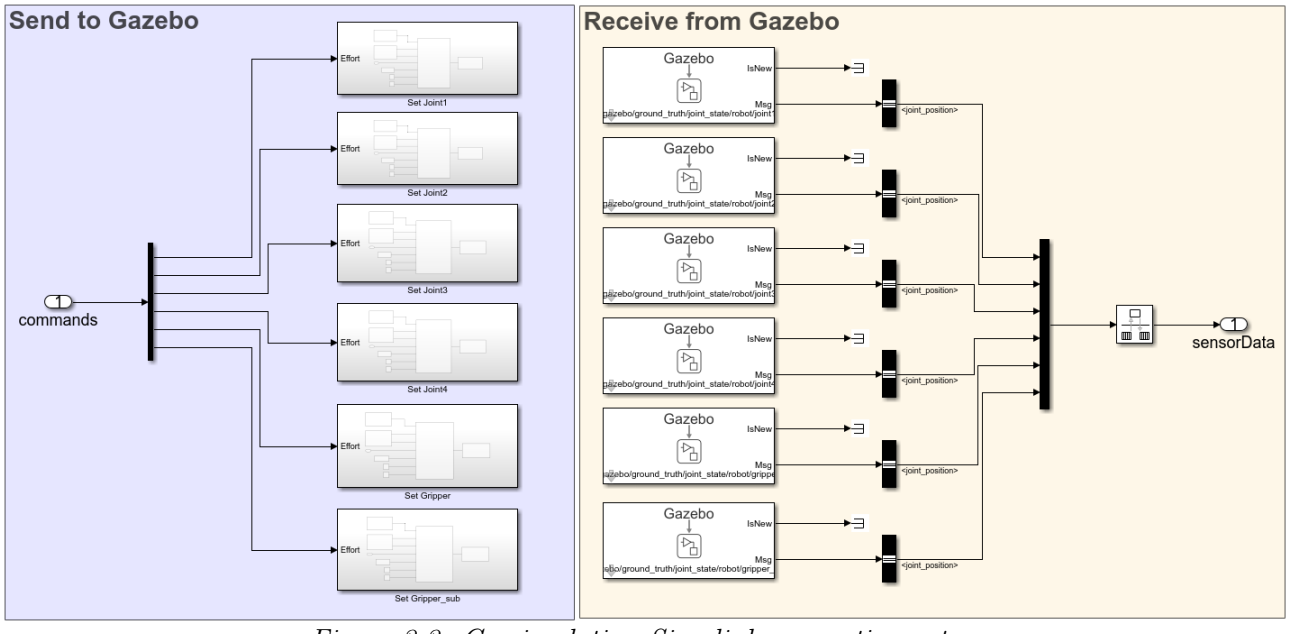


Figure 3.3: Co-simulation Simulink connection setup

4 Comparison Simscape vs Gazebo

In this chapter, the comparison between Simscape Multibody and Gazebo is made based on the criteria and the scenario defined in Chapter 2. First, the main characteristics of the environments and the general differences between the simulation workflows are compared. Then the sensor possibilities are elaborated and tested separately for each environment. Subsequently, the difference between joint limit and collision modelling is discussed after which the results of the time measurements are shown and evaluated.

4.1 General Characteristics

In Table 4.1, a general comparison between the main characteristics of Simscape Multibody and Gazebo is shown [32] [25]. (i) Looking at the simulator type, both environments are 3D simulators. Simscape Multibody is mainly used for mechanical system simulation while in Gazebo a more detailed environment around the robot can be simulated. (ii) Comparing them in terms of the supported operating system, Simscape Multibody works perfectly on Windows while for Gazebo it is preferred to run on a Linux system. Therefore a virtual machine is commonly used to run Gazebo. (iii) One advantage of Gazebo is the fact that it is open-source software, which means that no licence is required in contrast to Simscape Multibody where different licences are needed, such as for Matlab and Simulink. (iv) In terms of programming language, Simscape Multibody can be programmed mainly via Simulink blocks and Matlab functions. By using these pre-defined blocks, programming becomes relatively convenient since it gives a clear overview of the robot model. To simulate a robot in Gazebo, ROS is commonly used that consists of programming languages like C++ and Phyton. In addition, the Linux shell environment needs to be used which requires also a type of programming knowledge. Changing the robot model is done directly inside the URDF file of the robot. (v) In Gazebo, it is possible to use four different physics engines while in Simscape Multibody different solver methods can be chosen depending on the mechanical simulation scenario. (vi) MathWorks provides built-in functions that can import CAD models directly to the simulation environment. Different plugins allow CAD conversion to URDF, which can then be implemented in Gazebo. URDF and SDF models can be used for both environments but for Simscape a conversion is needed while in Gazebo it can be implemented directly. (vii) Finally, it is possible to combine the ROS environment with a simulation in Gazebo. The same codes that are used for Gazebo can be directly implemented on real hardware via this ROS integration. A ROS connection with Simulink can be established with the Robotic System Toolbox.

Table 4.1: General characteristics comparison

	Simscape Multibody	**Gazebo**
(i) Simulator type	3D Multibody Simulator	3D Robotics Simulator
(ii) Supported operating system	Mac, Windows, Linux	Linux/GNU (Ubuntu)
(iii) Licences	Licence for Matlab, Simulink, Simscape	Open-Source
(iv) Programming Language	Simulink, Matlab	C++, C, Python, Java
(v) Physics engine	Different solver methods	ODE, Bullet, Simbody, DART
(vi) CAD files / URDF support	URDF/SDF, STL, FBX, VRML, CITIA, DAE	URDF/SDF, STL, Collada, OBJ
(vii) ROS connection	Via Simulink or Matlab	ROS 1, ROS 2

4.2 Simulation Workflow

In this section, the general workflow is shown that is used to simulate the manipulator scenario in Simscape Multibody and Gazebo via co-simulation. These workflows are based on the test setup that is made for the comparison criteria. Therefore, they may differ, for example when a different scenario is simulated or when Gazebo is used without co-simulation.

A general workflow scheme that is used to set the manipulator scenario in Simscape Multibody and Gazebo is shown in Figure 4.1, showing the main differences. First, the URDF of the robot as well as the necessary STL files are needed for both simulation environments. For Simscape Multibody, the URDF first needs to be converted to the Simulink environment which can be done via a built-in function `smimport`. Next, the missing elements from this URDF that are ignored during the conversion need to be added, including the STL references to the corresponding visual elements of the robot body. In Section 4.5.1 the effects of these missing elements will be further elaborated. Finally, the robot model can be directly implemented in the Simulink file which contains the reference signals and controllers. Based on this workflow, it can be concluded that implementing a robot scenario in Simscape Multibody does not require many steps or prior programming knowledge.

For Gazebo the workflow is different. First, a new package is made in the catkin workspace of the virtual machine, after which the complete URDF of the robot can be added to the SDF/world file. Next, the necessary plugins can be added to the world or launch file, for example, the co-simulation plugin or ROS controller plugins if necessary. The catkin environment is then ready to build after which the environment can be opened with a before-created launch file. For the scenario, a co-simulation connection is used so that the controller is in the Simulink environment. Therefore, the last step is to set up the co-simulation connection within a Simulink file using the blocks of the Robotic System Toolbox as elaborated in Section 3.3. It can be concluded that for Gazebo basic knowledge is required about ROS (the catkin workspace and launch files), Linux shell environment and URDF/SDF files (XML language) which makes the setup relatively more complex compared to Simscape Multibody.

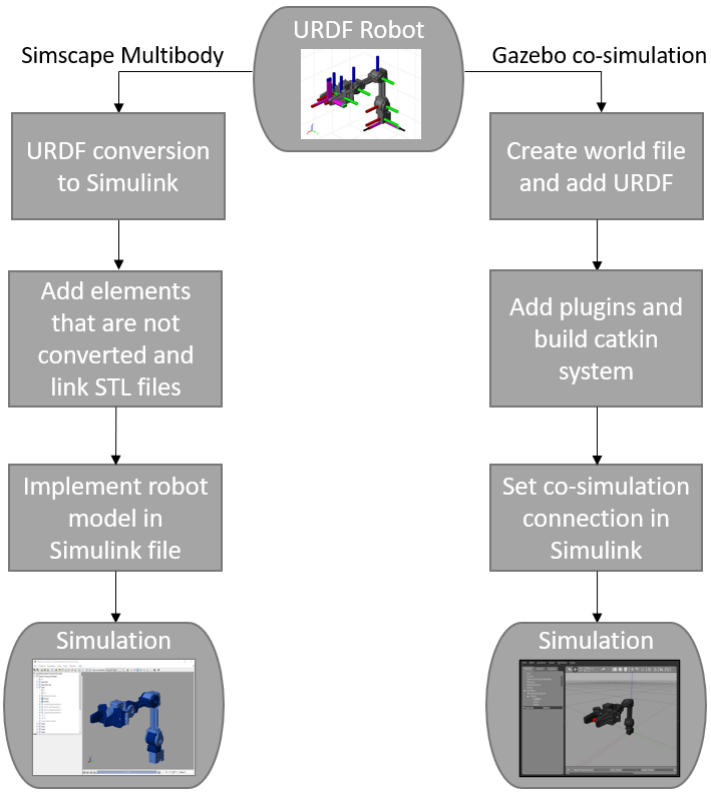


Figure 4.1: General simulation workflow for the two environments

4.3 Sensor Modelling in Simscape Multibody

In this section, the sensor simulation possibilities within Simscape Multibody are described including examples of built-in sensors that are demonstrated with the manipulator scenario. The Simulink files can be found in the repository [46].

According to the official MathWorks documentation [16] and previous literature [25], Simscape Multibody mainly focuses on the simulation of the mechanical and physical aspects of a robot. It provides limited functionality in terms of built-in sensor modelling possibilities that can generate synthetic data. No predefined virtual sensor models such as lidar, camera and IMU sensors were found that could be directly implemented without having to make an extra connection to another environment. A tool that requires an extra connection is the Simulink 3D Animation product. Via this tool, a 3D visualisation of the robot model can be simulated from the Simscape Multibody model [1]. Different sensors can be added to the scene such as a PointPickSensor. To use this tool together with Simscape Multibody, the complete visual representation of the robot needs to be made in the 3D Animation product. Also, an extra connection between Simscape Multibody blocks and this tool needs to be made in Simulink.

Using the Automated Driving Toolbox and Navigation Toolbox of Matlab, it is possible to model virtual lidar sensors. However, using these directly in Simscape Multibody is not feasible because this model is based on an Unreal Engine rendered environment. Co-simulation between Simulink and Unreal Engine is possible via the Automated Driving Toolbox, allowing to receive data from virtual sensors in Unreal Engine [15]. Physical models from Simscape Multibody can be implemented in Unreal Engine while controlling it via Simulink. Nowadays, this toolbox is mainly used for vehicle simulations but it is also possible for robotic applications. Furthermore, Matlab provides a Sensor Fusion and Tracking Toolbox that includes real-world sensor models. Nonetheless, no official documentation or example was found where these sensors are implemented in Simscape Multibody.

4.3.1 Transform Sensor

An example of a built-in sensor block from the Simscape Multibody toolbox is the Transform Sensor. This block measures the time-depended relationship between two specified frames. Giving the possibility to measure translation and rotational position, velocity and acceleration from the frames during the simulation. This sensor is commonly used in other robotic scenarios that are simulated in Simscape Multibody. For example, [39], where the sensor is used to directly measure the distance between the end effector of the ABB IRB360 robot and the table model.

The working of this sensor is demonstrated on the manipulator scenario to measure the distance between the specified world link and body link 5, see Figure 4.2 for the Simulink implementation. The input and output of the blocks are connected to the world frame and link 5 respectively. An additional output is enabled which sends the data during the simulation to the Matlab workspace. Between the Transform Sensor and the To Workspace block, a PS-Simulink Converter is used to connect Simscape physical network to Simulink blocks.

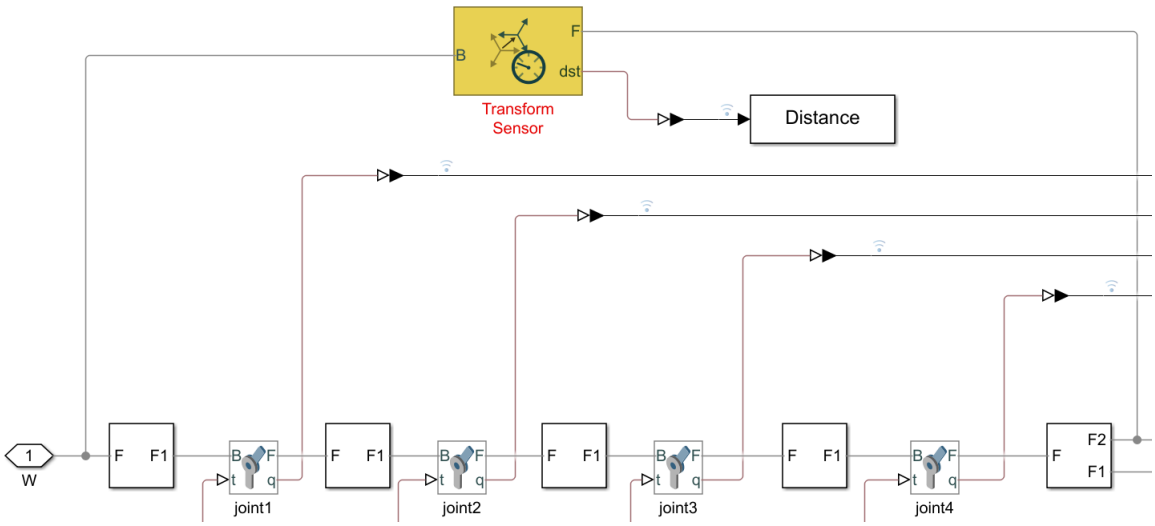


Figure 4.2: Transform Sensor block Simulink implementation

In Figure 4.3, the measured distance is plotted over the simulation time between the world frame and link 5 of the manipulator scenario. This sensor directly gives the exact distance between frames without noise, so there is no need to compensate for uncertainty.

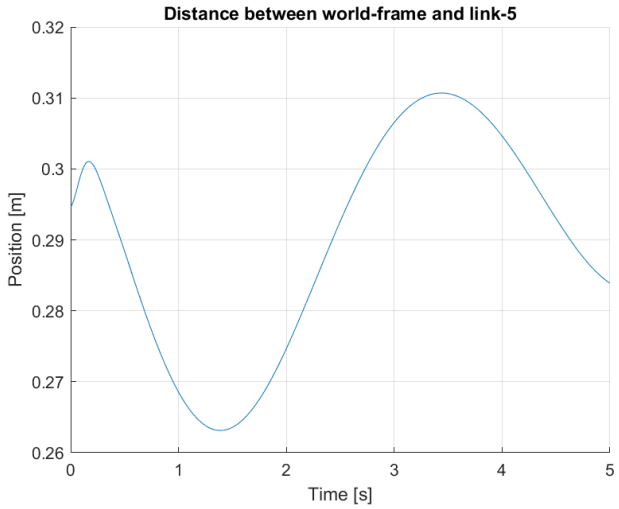
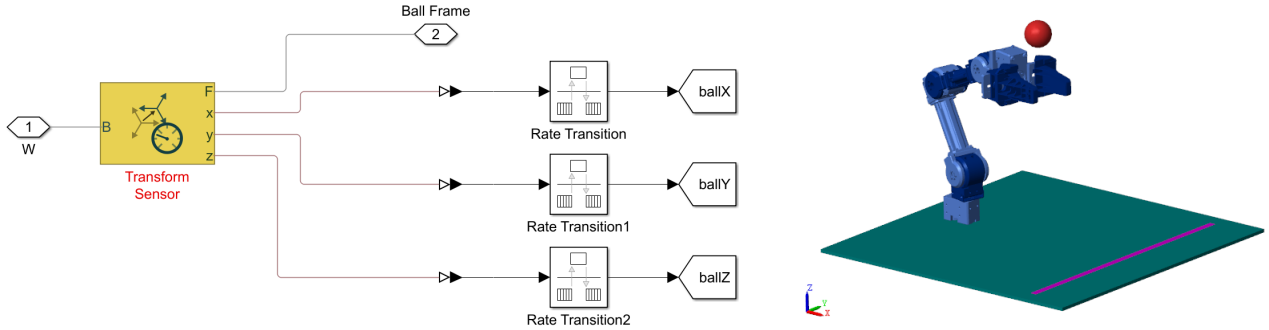


Figure 4.3: Transform sensor distance output plot

Since, for example, a virtual lidar sensor can not be modelled directly within the Simscape Multibody environment, the Transform Sensor block can be used to pretend/assume there is a sensor which can measure the distance to an object. This can be seen in the example shown in Figure 4.4. The OpenManipulator files from MathWorks [17] provide an example where the manipulator catches a ball falling. In this model, the sensor is used to measure the distance from the world frame to the ball. The measured distance from x, y and z is transferred to a state-flow chart where the reference position of the robot is calculated accordingly. The sensor makes it convenient to test controller algorithms without having to simulate complex synthetic data of virtual sensors since it directly gives the distance between the frames.

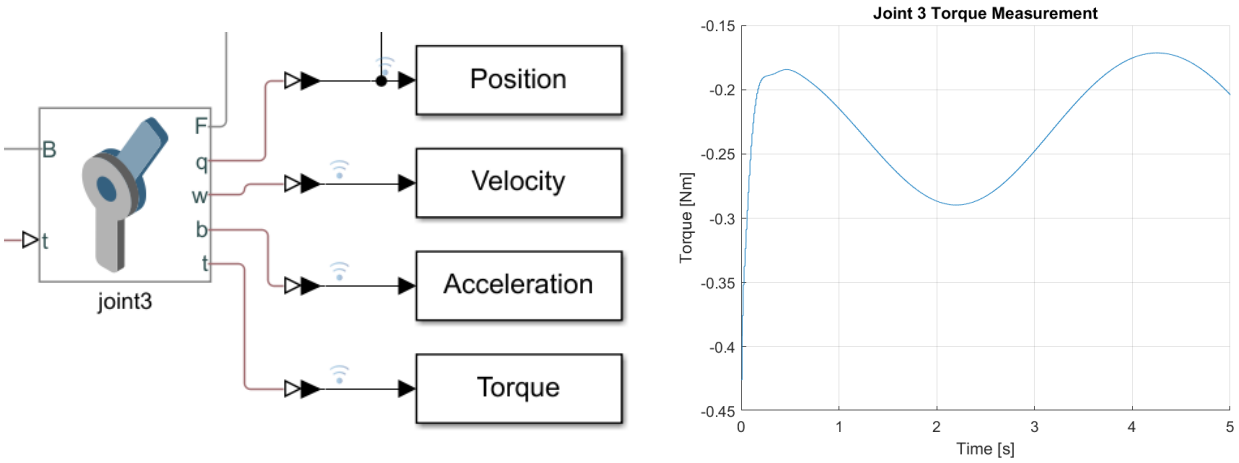


(a) Position camera sensor in Simulink *(b) Manipulator catching a ball*

Figure 4.4: Transform sensor application [17]

4.3.2 Joint Sensors

Within the revolute joint model block (see Figure 4.5a) of the OpenManipualator, Simscape Multibody provides built-in sensors that are able to directly measure joint position, velocity, acceleration and actuator torque during the simulation. These sensors can be enabled within the joint block whereupon the data can be stored in the Matlab workspace. In the manipulator scenario, the position of the joints is measured in this way and is then sent to the feedback controller. In Figure 4.5b, an example of the measured torque of joint 3 is shown during a simulation of 5 seconds.



(a) Built-in joint sensors implementation *(b) Torque data plot of simulation*

Figure 4.5: Joint sensors tested on scenario

4.3.3 Inertia Sensor

With the built-in time-dependent inertial sensor, it is possible to measure the mass and the centre of mass of one, or multiple specified body element(s). In addition, it can compute the inertia and rotation matrix of a specific link. This sensor is also included in the Simulink file of the repository [46]. Connecting the sensor to the manipulator in Simulink, the following information can be measured from the model:

- Mass of link
- Inertia matrix of link
- Rotation matrix of link

4.3.4 Simscape Ideal Sensors

Within the foundation library of Simscape, there are a lot of different sensor blocks. These are specified in different categorisations named: electrical, gas, hydraulic, magnetic, thermal, and mechanical. Sensors from all these categories can be combined in one Simulink model. This makes it possible to model the actuator of a revolute joint in more detail, taking the electrical domain into account [30].

The mechanical section contains four built-in sensors. These sensors are ideal, they do not take any inertia, friction, energy consumption and delays into account.

- Ideal Force Sensor
- Ideal Rotational Motion Sensor
- Ideal Torque Sensor
- Ideal Transnational Motion Sensor

These 1D sensors can be connected to the 3D Simscape Multibody environment to a single degree of freedom, for example, an actuator that operates in one direction. To make a connection between the 1D and the 3D environment, the Simscape Multibody Multiphysics Library provides blocks that establish this connection. This connection block makes use of these sensors. The library is also used in the OpenManipulator example to model a translational hard stop of the gripper, see repository [46] for the implementation. Within the 1D environment, friction and stiffness properties of the joints can then be simulated in more detail.

4.3.5 Conclusion on Scenario

➢ Simulating the OpenManipulator in the Simscape Multibody environment gives no possibilities to directly implement virtual sensor models like lidar, camera and IMU sensors without making an extra connection to environments. If the distance to an object within the environment needs to be measured, this can be done using the built-in transform sensor block, providing the distance, velocity, and acceleration between two frames. This gives the possibility to assume the distance is measured from a specific sensor without having to simulate synthetic data from a virtual sensor.

➢ Within the Simscape Multibody model of the manipulator scenario the joint position, velocity, acceleration, and torque can be measured directly from the joint blocks. This data, as well as other sensor data, can be extracted and stored during the simulation.

➢ It is possible to measure mass and inertia properties from selected bodies of the manipulator using the built-in inertia sensor block. Geometric properties, including the centre of mass and inertia matrix, can be measured from the complete manipulator model or specified subsystems.

➢ Built-in ideal sensors from Simscape give the possibility to model electric actuators of the joints of the manipulator scenario in more detail, making use of Simscape Electrical. 1D and 3D physical models from Simscape can be combined to extend the dynamical properties of individual joints.

4.4 Sensor Modelling in Gazebo

In this section, three types of sensors are evaluated and tested with the scenario that can be simulated in Gazebo: IMU, lidar and an RGB camera sensor. To visualise the data from the scenario, the sensor data is sent from Gazebo to Simulink via co-simulation. The Simulink files and the world files that are used for the setups are located in the repository [42].

Gazebo is generally well known for its large variety of sensor modelling according to other research on the comparison of simulation environments [36]. Based on other robotic scenarios in Gazebo, for example [33], complex environments for mobile robots with virtual synthetic sensor data are simulated. Within the Gazebo documentation, a variety of sensor classes are listed such as altimeter, camera, contact, GPS, IMU, lidar and magnetometer sensors [5]. Pre-defined sensor models in XML format can be added to the world file of the simulation environment together with the necessary plugins.

4.4.1 Lidar Sensor

A lidar sensor can measure the distance to the surface of an object using lasers. The sensor sends light impulses and then measures the time between the impulse being sent and reflected. This type of sensor is commonly used for mobile robots to map the environment and avoid obstacles. The Gazebo plugin from MathWorks (as elaborated in Section 3.3), provides code to model the lidar sensor. It includes a C++ code which transforms the sensor data from the simulation in real-time to Simulink. In Appendix C.2, the XML code used for the lidar is shown and is added to the world file of the scenario. In the first part of the code, a mesh of the Hokuyo is included which is needed to visualise the sensor in Gazebo. This mesh gives a visual real-world representation of the Hokuyo lidar sensor in the Gazebo environment. The second part of the code consists of the sensor model itself. A lot of different parameters can be set including the sample time, resolution, min/max angle, range and noise of the sensor.

The XML code of the sensor model is added to the world file of the scenario so that the sensor is simulated together with the OpenManipulator. Launching the world model gives the following results which can be seen in Figure 4.6.

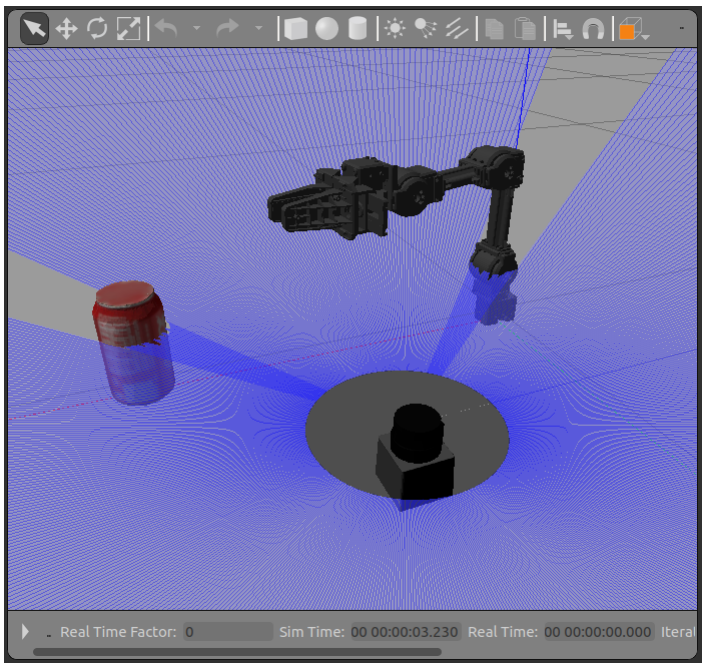
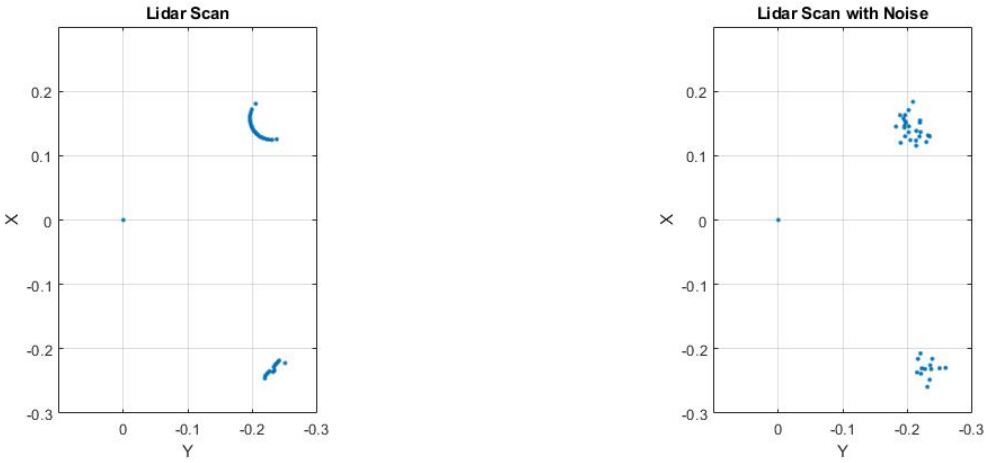


Figure 4.6: Lidar model with manipulator

Next to the manipulator, a Coke Can object is placed taken from the standard Gazebo world models. In the environment, blue lines can be seen which represent the laser signal of the lidar. The lidar laser is blocked by the two obstacles in the Gazebo environment, showing no laser contours behind the objects.

To collect the data modelled by the sensor, a new Gazebo Read block was added to the Simscape model in the same way as explained in Appendix A.3.2. The Gazebo topic of the Hokuyo sensor was selected to receive data from the sensor. Based on the MathWorks example [20], bus elements and a plot function were used to visualise the received data. In Figure 4.7, the data from the lidar is plotted for a specific time step of the simulation. Figure 4.7a shows the plotted data without a noise filter and Figure 4.7b shows the data after specifying a Gaussian filter in the XML code of the world file, simulating noise on the data.



(a) Lidar scan without noise *(b) Lidar scan with Gaussian filter*

Figure 4.7: Lidar plots from data received in Simulink

4.4.2 IMU Sensor

An IMU sensor is a device that consists of several sensors which can measure a variety of factors. It can measure the acceleration, speed, angular rate and magnetic field. The Gazebo plugin also provides the possibility to simulate an IMU sensor. In Appendix C.3, the XML code used for the IMU sensor is shown that has the same structure as the lidar sensor. To measure the acceleration, velocity and orientation of the OpenManipulator during the simulation, the IMU sensor needs to be attached to the arm of the robot. This is done by creating an extra link that specifies the position of the sensor with respect to the gripper link. The connection between the new link and the link of the robot is made by attaching an extra fixed joint in-between. The XML code for the created link and joint can also be found in Appendix C.3. In Figure 4.8, it can be seen how the IMU sensor is attached to the arm of the robot.

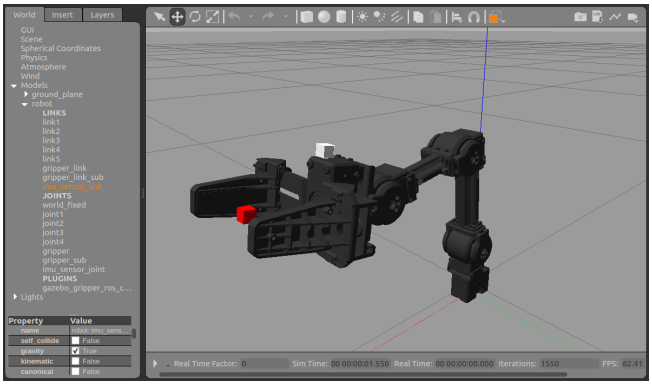


Figure 4.8: IMU sensor attached to the manipulator

During the simulation, the data from the IMU sensor is sent to Matlab via co-simulation. In Figure 4.9 the acceleration data measured in the Y-direction is plotted as an example. Likewise, the velocity and position of the IMU sensor can be stored in the Matlab workspace.

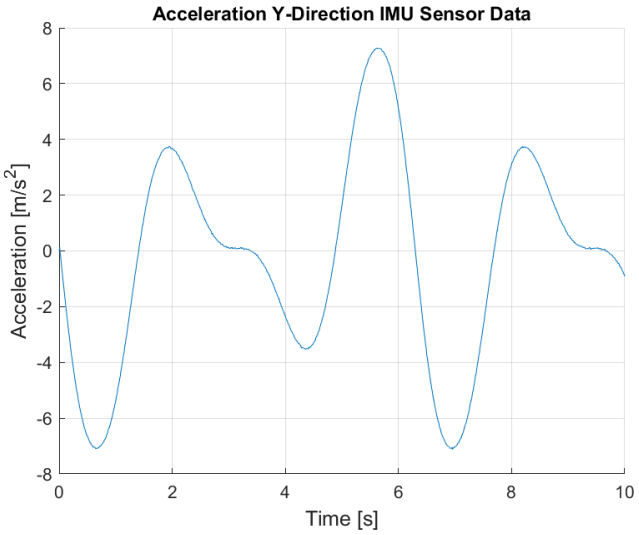
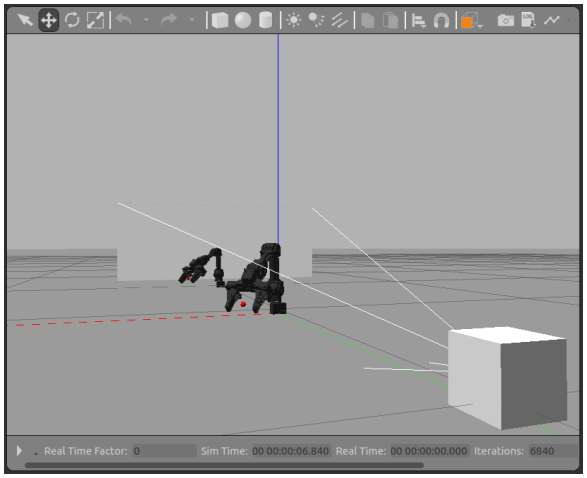


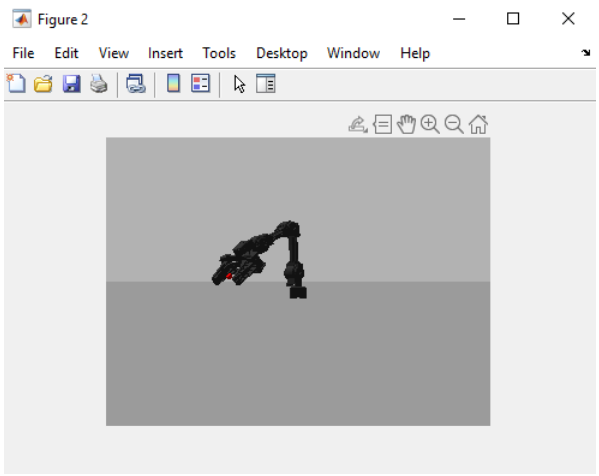
Figure 4.9: Plot of IMU sensor data during simulation

4.4.3 RGB Camera sensor

An RGB camera is a sensor that captures images representing the vision of the human eye in red, green, and blue wavelengths. This type of sensor can be used on robots to visualise the environment around them, making it possible to recognise objects based on their shape or colour. However, the depth cannot be measured with this type of sensor. As the same as for the other sensors, this sensor model is also included in the Matlab plugin and can therefore be added to the SDF file directly. The XML code is added to the world file in the same manner as for the other sensors. Figure 4.10a shows how the camera model is visualised in the Gazebo environment. The lines represent the field of view which can be changed in the XML code. Behind the robot, the captured vision of the camera sensor is shown that updates during the simulation.



(a) RGB camera in Gazebo



(b) Camera plot in Matlab

Figure 4.10: RGB camera simulation

The data from the camera is transferred to Matlab where it can be visualised, see Figure 4.10b. In the Simulink model, it is possible to increase or decrease the sample time of the camera data that is sent to Simulink. The RGB camera can also be mounted on top of the robot the same way as was done for the IMU sensor, see Section 4.4.2. In Gazebo, the camera projection and the white lines move along with the robot's movement. In Figure 4.11 it can be seen how the camera is attached to the gripper of the robot.

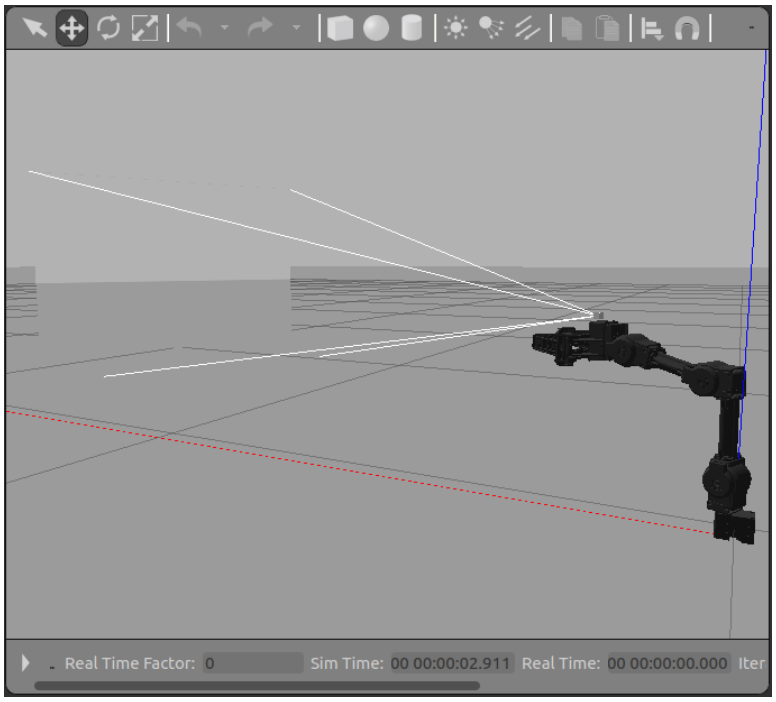


Figure 4.11: RGB camera model attached to gripper link

4.4.4 Conclusion on Scenario

- ➢ Simulating the manipulator scenario in the Gazebo environment provides the possibility to simulate virtual sensors, such as lidar, camera and IMU sensors, including synthetic sensor data. This data can be sent to Simulink during the simulation using the co-simulation connection.
- ➢ A lidar sensor can be simulated within Gazebo, measuring the distance to obstacles within the environment. This information could be used to approximate the location of an object next to the manipulator.
- ➢ Internal measurement units, such as an IMU sensor, can be connected to a link of the manipulator. The orientation, speed, and acceleration of the connecting link can be measured during the simulation.
- ➢ Data from an RGB camera model can be visualised in Matlab during the simulation. The camera can be attached to the robot so that the viewpoint of the camera changes accordingly. For example, the synthetic data could be further used for algorithms that can recognise the colours or shapes of objects. This allows the manipulator to distinguish between different colours and objects within the environment.
- ➢ In the SDF file, many settings of the sensors can be set. It is possible to simulate sensor noise by applying a Gaussian filter on the sensor output data before sending the information to Simulink. More options such as weight, visual description, range and the sample rate of the sensor can be modified or added.

4.5 Joint Limits and Collision Modelling

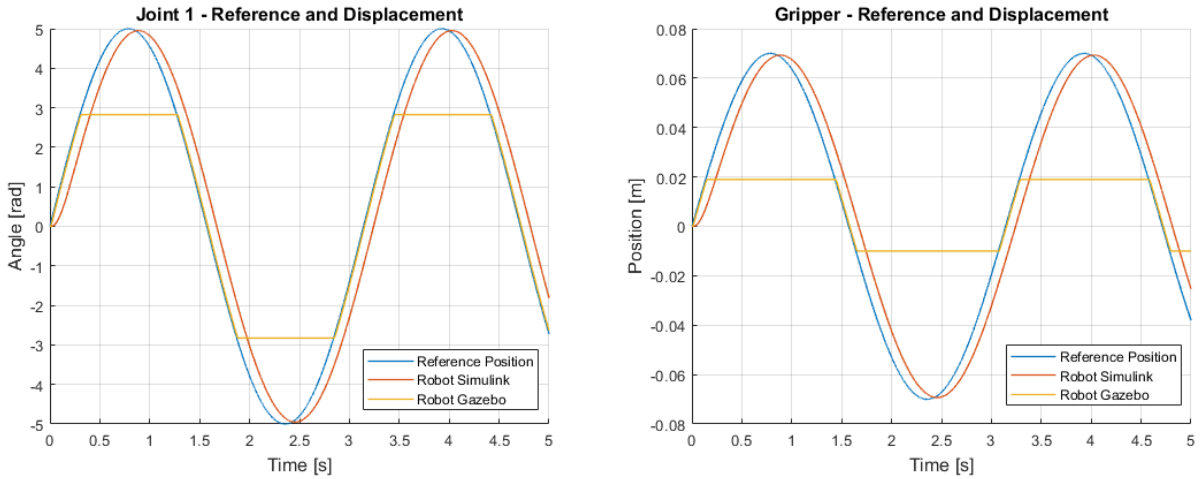
In this section, different tests are done with the OpenManipulator to investigate the differences between the simulation in Simscape Multibody and Gazebo in terms of joint limits and collision modelling. The underlying reasons for these differences have been further investigated and described.

4.5.1 Joint Limits

The OpenManipulator consists of four revolute joints and two prismatic joints. These joints have a limited range of motion defined by "joint limits". For the revolute joints, this range is defined by an angle (in rad or deg) and for the prismatic joints the limit is defined by a distance (in meters). The following measurement shows the differences between the trajectory of the robot based on a position as a reference signal. This can be used to see what differences there are based on simulating joint limits that are defined in the URDF of the robot. The files used for these measurements can be found in the repository [45].

Testing Joint Limits in Environments

To test the trajectory differences, a sinus function with an amplitude of 5 and 0.07 was set as input for joint 1 and the gripper joint respectively. In Figures 4.12a and 4.12b it can be seen that for the OpenManipulator in Simscape Multibody, no joint limits are simulated in contrast to the model in Gazebo. This difference can also be seen in the 3D visualisation, where the gripper seems to have no limit, see Figure 4.13.



(a) Joint 1 - Comparison plot *(b) Gripper 1 - Comparison plot*

Figure 4.12: Joint limit comparison plots

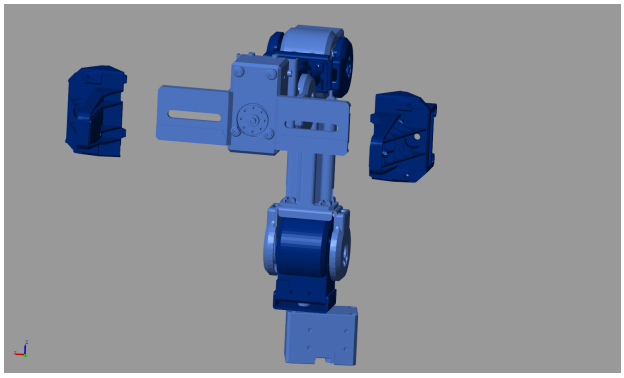


Figure 4.13: Gripper joint limit test in Simscape Multibody

Reason for Joint Limit Differences

Despite the fact that for both simulation environments the same URDF model of the manipulator is used, Simscape Multibody does not include the joint limits defined in this model. Although it is possible to simulate the joint limits in this environment, they are not simulated directly. To find a reason for this difference, further investigation is done regarding the URDF implementation between Simscape Multibody and Gazebo. Based on the support documentation of MathWorks about URDF import [22], it was found that there are limitations when importing a URDF into Simscape Multibody since not all elements from the URDF will be converted. This means that the robot model behaves differently compared to the model in Gazebo where these elements of the URDF description are taken into account. Below, a list of attributes is shown which are ignored when importing the URDF of a manipulator to the Simscape Multibody environment:

- <**transmission**> Defining a relationship between the actuator and joint of the robot to model gear ratios.
- <**gazebo**> Defining simulation properties for the Gazebo environment.
- <**model_state**> Setting "home" position in URDF models.
- <**sensor**> Defining sensor model and settings.
- <**collision**> Describing collision shape and parameters.
- <**limit**> Joint motion limits.
- <**scale**> Scaling the mesh of a body.
- <**friction**> Friction in joint internal mechanics.
- <**geometry**> Creating boxes and spheres.

Comparing this list with the URDF model of the OpenManipulator, the following elements are not taking into account by the conversion: <collision>, <limit>, <scale>, <friction> and <geometry>. This means that the standard model of the robot in Simscape Multibody does not automatically take collision modelling, joint limits and friction of the contact surfaces into account. This explains why there was a difference between simulating joint limits between the environments while using the same URDF of the robot.

Simulation Joint Limits in Simscape

Even though the joint limits are not directly taken into account, it is possible to include them in the Simscape Multibody simulation. Joint limits can be set inside the revolute joint block of the model. When implementing these limits, it was found that joint limits only work when the input is torque (so no position input). In Figure 4.14, it can be seen what effect the joint limits have on the trajectory of OpenManipluator, using the torque Simulink model as explained in Section 2.2.

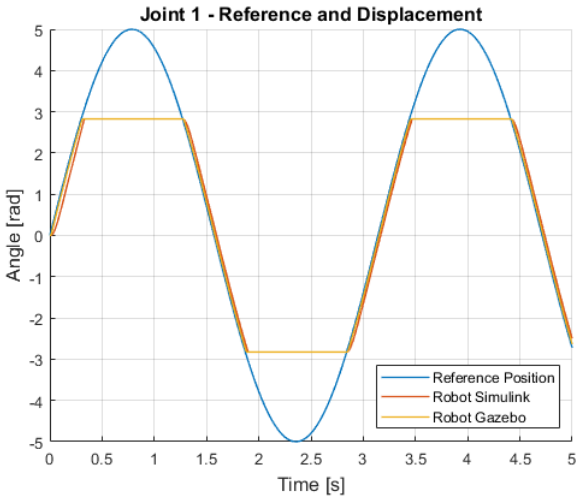


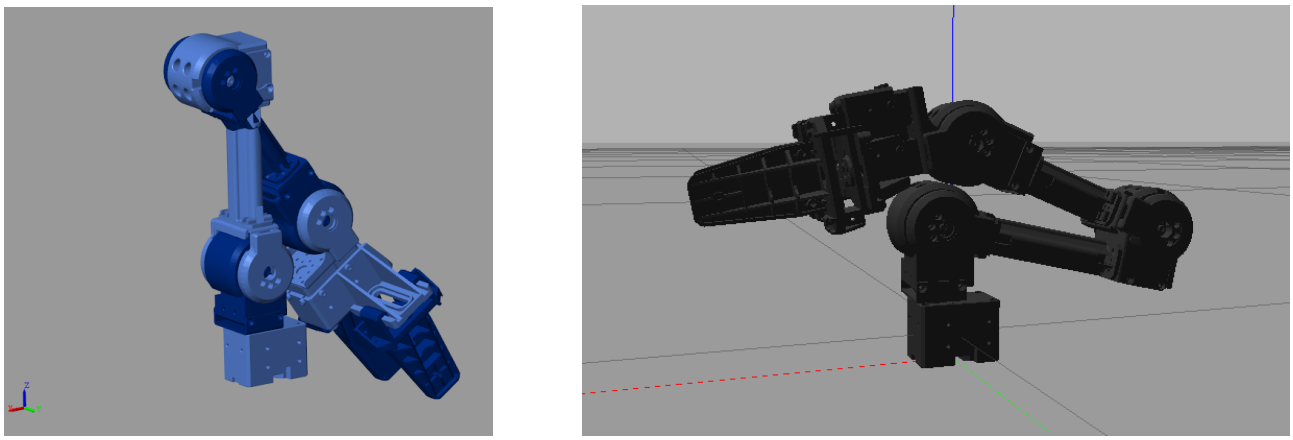
Figure 4.14: Joint limit simulation for both environments

4.5.2 Contact Modelling

When it comes to robotics simulation, modelling contacts is one of the most difficult tasks. Bodies from a robot can move in complicated ways and can have complicated geometries. Contact modelling is important when the robot needs to interact with the environment. In the following experiments, the differences between the OpenManipluator in Simscape Multibody and Gazebo are tested concerning environment and self-collision. The corresponding Simulink files are stored in the repository [44].

Testing Self and Ground Collision

First, it was tested if the manipulator scenario includes self-collision in Simscape Multibody and Gazebo. In Figure 4.15a, it can be seen that in the Simscape Multibody environment no self-collision is simulated. The manipulator can move through itself without any interaction between the bodies. In contrast to Simscape Multibody, it can be seen in Figure 4.15b that the manipulator in Gazebo is blocked by its own body. The contact boundaries of the self-collision correspond with the geometry of the manipulator. As explained in 4.5.1, the collision element from the URDF of the OpenManipulator is ignored during the conversion to Simscape which is in line with the results of the experiment.

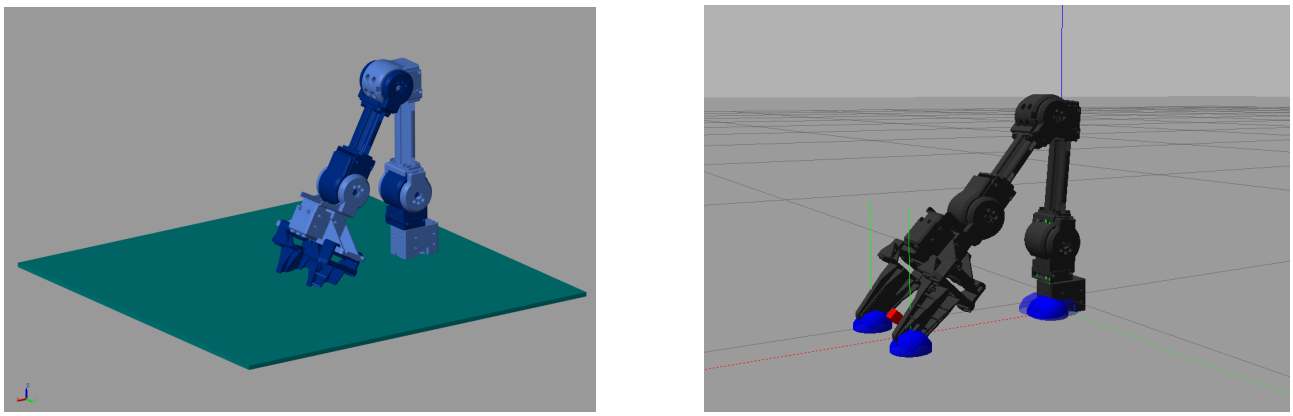


(a) Simscape Multibody *(b) Gazebo*

Figure 4.15: OpenManipulator self-collision test

In the following experiment, the collision was tested with the ground of the environment below the OpenManipulator. For the Gazebo simulation, a standard ground plane was included in the SDF

file and for Simscape Multibody an extra plane was added (provided by the MathsWork example [17]) to the model of the robot. As expected, Simscape Multibody does not automatically simulate collision interaction with the environment, see Figure 4.16a. In Gazebo, interaction with the ground was simulated as can be seen in Figure 4.16b. The blue balls indicate the collision contacts and the green lines show an active force vector. It can be seen that the collision boundary is equal to the geometry of the robot itself.



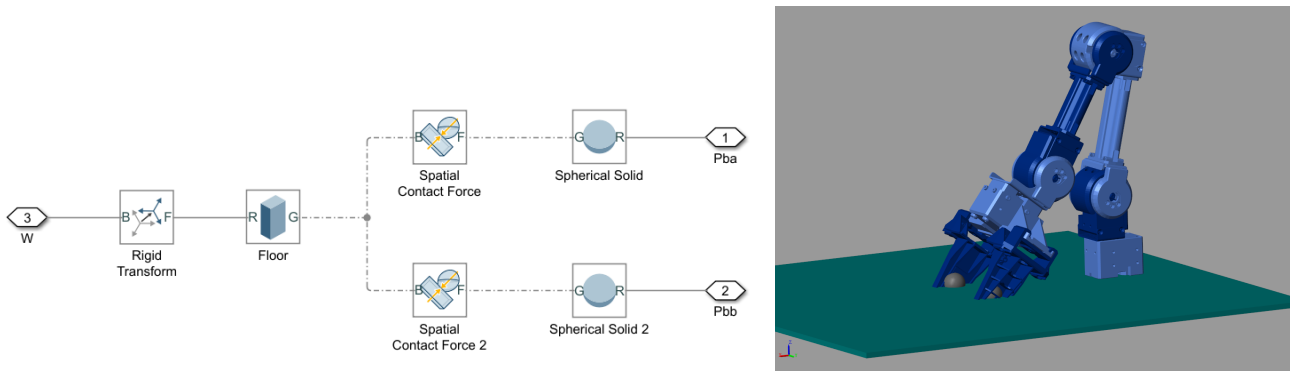
(a) Simscape Multibody *(b) Gazebo*

Figure 4.16: OpenManipulator ground collision test

Collision Modelling in Simscape Multibody

To explain the differences, further investigation is done about collision modelling in Simcape Multibody. In general, Simscape Multibody does not support modelling collision interaction between complex shapes, such as the geometry of the manipulator itself [29]. Modelling contact forces is mostly done with simple shapes such as spheres, cylinders and planes. Instead of simulating the complete geometry of an object as a collision element, spheres can be added as contact points. The Simscape Multibody Contact Forces Library includes pre-defined contact force models for these simple shapes including 2D and 3D problems.

The following example shows how a sphere can be used as a contact boundary to simulate a collision between the OpenManipulator and the ground plane. In Simulink, a connection between the ground floor and the spherical solids is made using the Spatial Contact Force block, see Figure 4.17a. In this block, the contact stiffness, contact damping and transition region width are defined. Doing the previous experiment again, it can be seen that the OpenManipulator cannot go through the ground plane because the physical interaction between the spheres and the ground plane is simulated, see Figure 4.17b.



(a) Contact spheres implementation in Simulink *(b) Contact spheres simulation*

Figure 4.17: Contact spheres tested with OpenManipulator

During the collision in the example of the robot, Simscape Multibody makes use of a penalty method [23]. The collision objects are simulated as a stiff spring with damping that is only enabled when the bodies are in contact with each other. This allows the sphere to penetrate the ground for a small amount. During this collision, the normal forces are computed according to the spring-damper force law. The more the objects penetrate each other, the greater the normal force. Within the Spatial Contact Force block, the normal force and friction force magnitude can be measured during the simulation.

Collision Modelling in Gazebo

In the Gazebo simulation environment, the collision contact geometry is equal to the mesh of the link as was shown in Figure 4.16. This collision boundary is defined in the URDF of the model underneath the collision element. An example of this code can be seen in Appendix C.5, where the collision boundary is defined for the first link of the OpenManipulator.

To understand how Gazebo simulates collisions, it is necessary to take a closer look at the physics engine used by Gazebo. For the manipulator scenario, the Open Dynamics Engine (ODE) is used as specified in the SDF model. A physics engine is computer software that calculates the dynamical behaviour including body collision, friction and joint behaviour during the simulation. The ODE uses "hard contacts" to simulate a collision between objects. This means that when these objects collide with a given velocity, a non-penetration constraint is used [11]. It is not possible to penetrate the surface and therefore the contact force does not vary over time. This means that the "true" contact time is almost zero. The effect of the collision is simulated by giving the objects post-collision movement by a momentum exchange. This method is commonly used for real-time physics engines, where computation speed and robustness are key. Other physics engines make use of a spring contact (soft contacts), where penetration is possible as with Simscape Multibody. These soft contacts can be used to simulate real contact forces, but are computationally expensive and more prone to errors.

4.5.3 Conclusion on Scenario

- When a URDF from a robot is imported into Simscape Multibody for simulation, not all elements are included, for example, joint limits. However, this can be implemented in the Simscape Multibody model after the conversion. In Gazebo, all elements of the URDF are taken into account which makes it possible to simulate the joint limits directly.
- The specified collision boundaries in the URDF of the OpenManipulator are not taken into account in Simscape Multibody. The robot can move through itself and the ground plane without any physical interaction. This is different in Gazebo, where the robot collides with itself and the ground plane.
- Collisions in Simscape Multibody are mostly simulated by simple objects like spheres. These spheres can be placed on objects that need physical interaction. Reaction forces are computed during the collision by simulating the bodies as virtual springs.
- The ODE that is used for the scenario in Gazebo simulates collisions using the hard contacts method, which is faster but not as accurate as soft contacts. The bodies are not simulated as virtual springs resulting in the contact force being constant during the collision.

4.6 Simulation Time

In this section, the differences between the simulation setup of the robot in Simscape Multibody and Gazebo are tested in terms of simulation time, using the position as reference input. In addition, the impact when adding a sensor to Gazebo will be tested with respect to the simulation time. The definition of simulation time is the actual time needed to complete a simulation of a given robot model. This time can be influenced by a lot of different factors, such as computer specifications, model complexity and the computational efficiency of the simulation software. Therefore it is important to mention that these results only hold for these particular Simulink setups for the chosen scenario on a particular computer. The setup conditions are shown in Appendix A.5. The virtual machine including the world files and the Simulink models are stored in the repository [47].

4.6.1 Measurement Method

The simulation time is measured within Simulink using the built-in "profiler" tool for both simulation environments. This tool can measure the time it takes to complete a simulation in Simulink. After each simulation, the tool gives the total simulation time for three decimal places including the time it takes for different blocks in the model to be simulated. Only the total simulation time is considered for the measurements. It is important to mention that for Gazebo, the simulation time is also measured from the Simulink model since co-simulation is used. Thus, not only is the time measured that the environment takes to simulate, but the entire simulation time, including the Simulink model with the co-simulation connection to the virtual machine. In Appendix A.5, an example measurement using the profiler tool is shown.

For the measurements with Simscape Multibody, a fixed-step continuous implicit solver is used, named ode14x. This solver is chosen because it is a fixed-step solver and recommend for physical models that are stiff. Accordingly, the step sizes for the simulations are changed in the solver settings. For the Gazebo measurements, the standard ODE (Open Dynamics Engine) is used for the simulations. The step sizes are changed in the pacer block which is located in the Simulink model as shown in Section 3.3.2.

The same reference signals are used for both Simulink models, containing sine waves for each of the four revolute joints and a step signal for the gripper. The stop time in Simulink was set to 5 seconds for all the measurements. To minimise the influence of various factors during the time measurements, the simulations are carried out on one computer with no programs running in the background. See Appendix A.5 for the computer and virtual machine specifications. Each simulation is carried out 10 times from which the average, standard deviation and real-time factor (ratio between the time taken for the simulation and the input duration) is calculated.

The simulation time is compared in two different ways. First, the simulation time is compared between Simscape Multibody and Gazebo using a Simulink model with a position as a reference signal. The sample time of the simulation is changed from 0.01 to 0.001 seconds. Secondly, the simulation time of Gazebo is compared for different simulation step sizes together with an RGB camera sensor mounted on the robot, shown in Section 4.4.3. The step size for this camera is varied from 0.1 to 0.01 seconds.

4.6.2 Simulation Time Results

In Table 4.2 the results for the four different measurements are shown including the main settings for the different simulation typologies. The results for the mean simulation time of the environments with different step-sized are visualised in a bar chart as can be seen in Figure 4.18.

Table 4.2: Simscape and Gazebo simulation time measurements

Simulator	Simulation step-size	Solver method	Mean simulation time	Sample standard deviation	Real-time factor
Simscape	0.01	ode14x	0.417	0.031	0.083
Gazebo	0.01	ODE	11.649	0.176	2.330
Simscape	0.001	ode14x	2.375	0.024	0.475
Gazebo	0.001	ODE	51.550	0.522	10.310

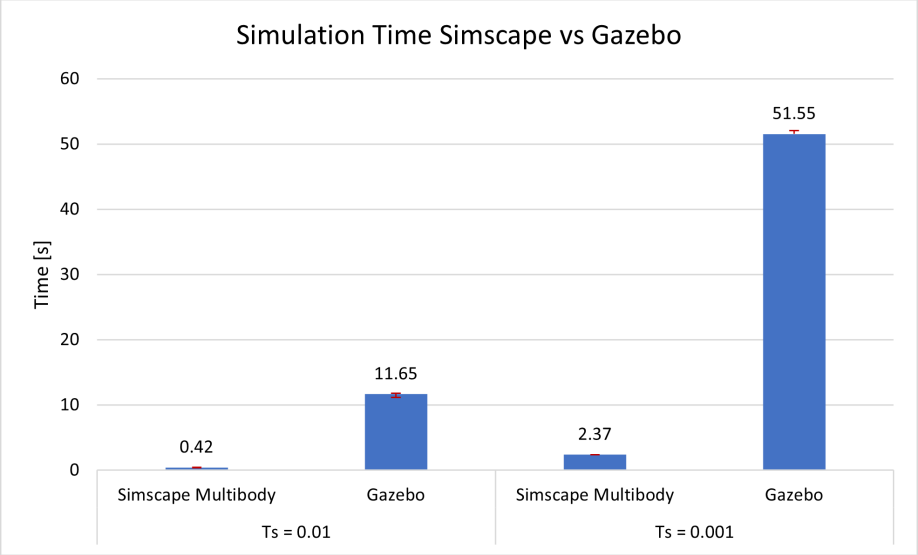


Figure 4.18: Simulation time Simscape vs Gazebo

It can be seen that in general Simscape Multibody is considerably faster than Gazebo for both step sizes. For a step size of 0.01 seconds, Simscape is around 28 times faster than Gazebo. Having a step size of 0.001 seconds, Simscape is around 22 times faster. The simulation times of Simscape are below 5 seconds and therefore faster than real-time, in contrast to Gazebo which is around 2 and 10 times slower than real-time. In addition, it can be seen that the sample standard deviation is higher for Gazebo than for Simscape, which means that Gazebo shows more variation in the simulation time. For both environments, the variation increased when lowering the simulation step size.

The results for the second comparison are shown in Table 4.3 whereupon the mean simulation time and the standard deviation are visualised with a bar chart in Figure 4.19.

Table 4.3: Gazebo sensor simulation time measurements

Simulator	Simulation step-size	Camera step-size	Mean simulation time	Sample standard deviation	Real-time factor
Gazebo	0.01	none	11.649	0.176	2.330
Gazebo	0.001	none	51.550	0.522	10.310
Gazebo	0.01	0.1	14.733	0.450	2.947
Gazebo	0.01	0.01	28.546	1.227	5.709
Gazebo	0.001	0.1	63.646	2.479	12.729
Gazebo	0.001	0.01	81.787	3.505	16.357

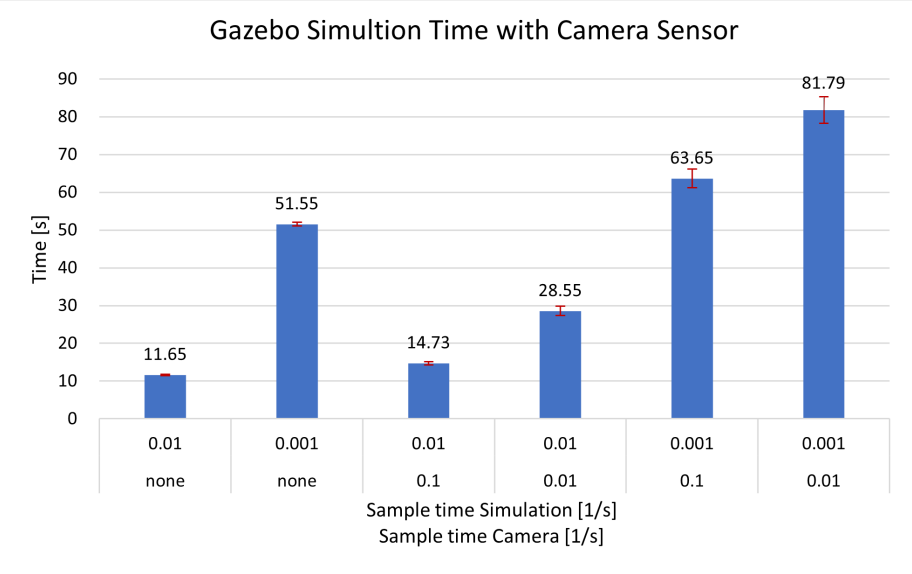


Figure 4.19: Simulation time of RGB sensor in Gazebo

From these results, it can be seen that the simulation time increases when adding an RGB camera sensor to the robot. Adding this sensor can increase the time by around 3 to 30 seconds depending on the sample times. In addition, the sample standard deviation is higher for the simulations that include a camera sensor. Comparing the results with a sample time of 0.001 without a camera and 0.001 with a camera (0.01), the sample standard deviation increases by 571.5 per cent.

4.6.3 Conclusion on Scenario

- For this particular setup, the simulation of the manipulator scenario is considerably faster in Simscape Multibody than in Gazebo with co-simulation. In Simscape Multibody the simulation is even faster than in real-time, while in Gazebo it takes at least double the amount of time. It is important to mention that these simulations only use a sinus signal as an input reference. The simulation time results could be completely different when, for example, collisions between objects are simulated or when a different solver method is used. The measurement results show that despite the relatively simple robot, real-time is difficult to achieve with this co-simulation setup.
- The sample standard deviation of the Gazebo measurements is larger for all cases compared to Simscape Multibody, making these simulations less consistent.
- The implementation of a camera sensor has a major effect on the simulation time when relatively low sample size is chosen. This makes this co-simulation setup, given the simulation speed, not the preferred method to simulate multiple sensors with a low sample time.

5 Perception in Gazebo

In this chapter, based on a use case defined by a graduation project, sensor simulations in Gazebo are further explored and applied. The chapter starts with a description of this use case, followed by an explanation of the test setup in Gazebo. Next, it is shown how a 2D and 3D grid map can be made based on sensor simulation in Gazebo. Finally, the 3D grid map is tested on a simulation of an Image-guided Therapy robot from the use case. Files used for the perception in Gazebo can be found in the repository [41].

5.1 Use Case Description

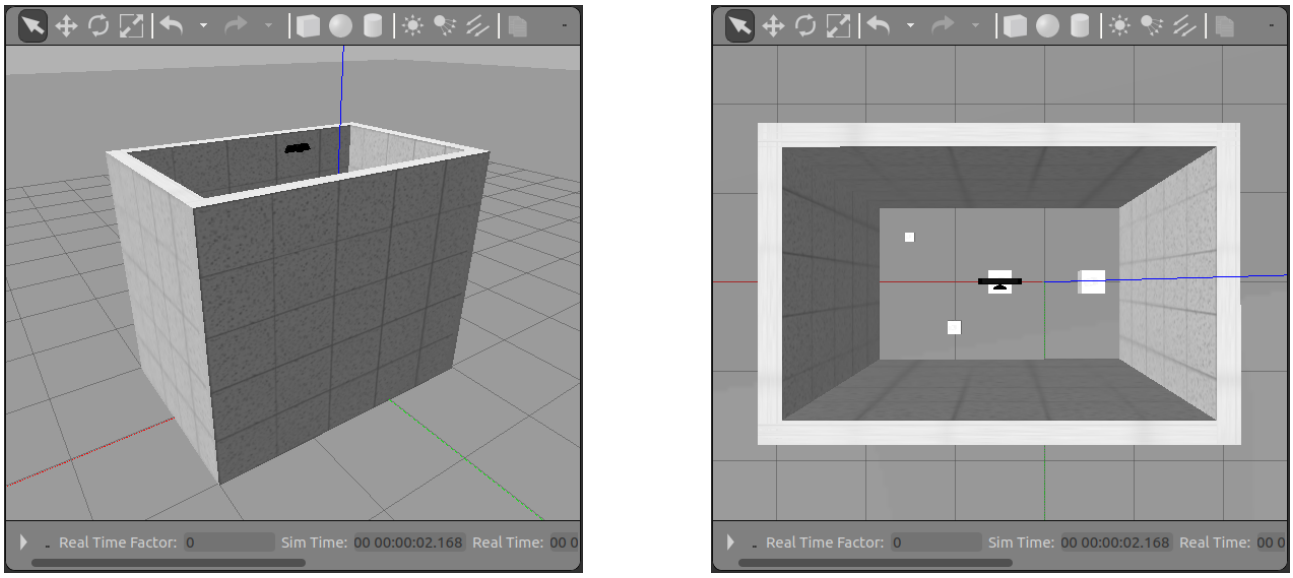
The investigation of sensor possibilities in Gazebo is further elaborated using a use case from a master's thesis. In this thesis, a controller is being developed for an IGT robot. This robot should move from point A to point B without colliding with static and dynamic obstacles in the room. For obstacle avoidance, the controller needs to know the distance between the robot's body and the obstacles. In order to measure this distance, a 2D or 3D top-view map of the room needs to be made. These maps are called "binary occupancy grid maps", they consist of a grid that represents the location of obstacles. These grids are equal to 1 for occupied, and 0 for empty spaces. For dynamic obstacles present in the room, it is important that the grid map can be updated in real-time during the simulation. Since the controller is working within Matlab, the sensor information from Gazebo needs to be transferred to the Windows environment where Matlab is located. Based on the sensor comparison as described in Chapter 4, it can be concluded that Gazebo gives the possibility to co-simulate with Simulink. Moreover, it is able to model virtual sensors and transform this data during the simulation to Matlab. Since these requirements are necessary for the use case, Gazebo was chosen as the simulation environment.

5.2 Gazebo Room with Depth Camera

To investigate the depth sensor configuration and data transfer to Matlab, a test setup in Gazebo was made consisting of a room with four walls. These walls were implemented using the Gazebo Building Editor tool. This tool automatically generates 3D walls in the Gazebo simulation world based on the dimensions specified in the 2D view of the build editor. A room of 2 by 3 meters was created with this tool including four objects placed on the floor of the room. These objects in the shape of boxes, together with the walls, are specified inside the SDF file containing the dimensions and positions.

For the setup, a depth camera is needed which is mounted at the top of the room, this ensures that a big part of the room can be captured. The type of depth camera that is used for this setup is the Microsoft Kinect sensor. The sensor consists of a 3D depth sensor and a normal RGB camera. Combining these gives the possibility to measure the depth signals simultaneously with the RGB images. The depth sensors use an IR laser projector together with an IR camera, giving the possibility to create a 3D map with a resolution of 640 x 480 pixels at 30 Hz. The RGB camera captures images with a resolution of 640 x 480 pixels at 30 Hz but can be increased to 1280 x 1024 pixels running at 10 Hz. The Kinect sensor is a commonly used sensor for obstacle avoidance as shown in research [13]. This research concluded that the Kinect sensor is best used in indoor environments because there the IR absorption is much lower than in outdoor environments. Moreover, the sensor has a limited range detection, between 0.5 and 6 meters. In the use case, the sensor is mounted in an indoor environment and does not need a range larger than approximately 5 meters. Therefore, the Kinect sensor is adequate for the simulation.

To simulate the Kinect sensor in Gazebo, the depth camera ROS plugin is used (included in the Gazebo ROS package [6]). This plugin gives the possibility to simulate depth sensors and provides a ROS interface, which allows for the publishing of the data from the Kinect sensor via ROS messages. The Kinect sensor is added to the SDF file of the previously created room. To mount the sensor at the top of the room, a fixed joint is made that is attached to the world frame and also to the link of the camera sensor. The XML code of the camera model including the created joint can be found in [41]. The setup in Gazebo with the four walls, boxes and Kinect camera can be seen in Figure 5.1.



(a) Side view of Gazebo room *(b) Top view of Gazebo room*

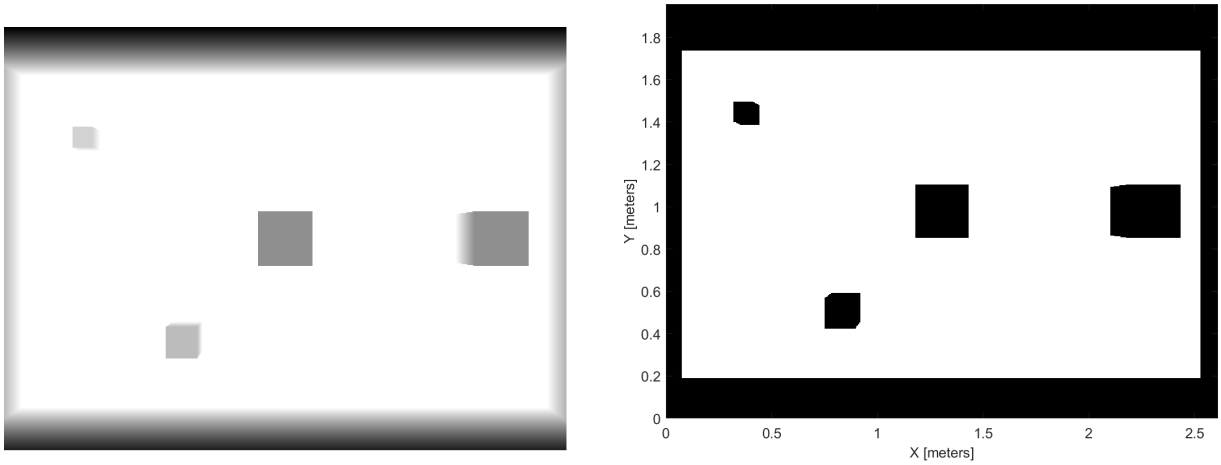
Figure 5.1: Gazebo test room with Kinect sensor and obstacles

The following camera ROS topics are available when starting the simulation and will be further used in the next sections for 2D and 3D grid maps:

- *camera/depth/colour/image_raw*
- *camera/depth/image_raw*
- *camera/depth/points*

5.3 2D Binary Occupancy Map

The camera ROS topics are further used to create a 2D grid map. Via the rqt_image_view package [28], a depth image can be visualised subscribing to the *camera/depth/image_raw* ROS topic. This image shows the depth of the objects captured by the Kinect camera, see Figure 5.2a. This image can be saved in PNG format, after which it can be loaded into Matlab. Accordingly, the `binaryOccupancyMap` function from Matlab is used to automatically create a binary 2D occupancy map. The PNG image is first converted to a black and white image before it can be used with the occupancy map function. In addition, the function requires a resolution specified as the number of pixels in one meter. To estimate this, the "spatial_calibartion_demo.m" script is used from the MathWorks forum. This script measures the length of objects and calculates the number of pixels selected after inserting the real length. In Figure 5.2b, the final result of the 2D grid map is shown. The Matlab script used for this map can be found in Appendix A. Having this grid map, the `checkOccupancy` command can be used to check if a certain specified position (in meters) is occupied by an obstacle or empty.



(a) Camera depth PNG-image *(b) 2D Binary Occupancy Map*

Figure 5.2: Camera depth image and 2D Binary Occupancy Map

5.3.1 Real Time Updating to Matlab

For the use case, it is also important to display dynamic objects in the grid map. Therefore, it was further investigated to update the 2D grid map in real-time to Matlab. By connecting with Matlab to the ROS master of the virtual machine, it is possible to subscribe to ROS topics and receive the information that is published. In this way, information from the */camera/depth/image* topic can be transferred to Matlab for further processing. The Franka Panda Robot was added to the test room in Gazebo, including a co-simulation connection between Simulink and Gazebo in the same way as described in Chapter 3. By sending a reference signal to the robot via Simulink, it could be further tested how dynamic objects can be registered in a 2D grid map. The Matlab scripts used for the gird map generation and the ROS connection can be found in Appendix B.2. The rate at which the 2D grid map is updated in Matlab is measured to be 0.186 to 0.283 seconds, which is fast enough to update the displacements of the Panda robot in the 2D grid map.

5.4 3D Occupancy Map via OctoMap Package

Having a 3D grid of the room gives a more detailed representation of the room compared to a 2D grid map, for example, the height of the obstacles. Therefore, it was further investigated how a 3D map can be generated for the Gazebo simulated point clouds and how it can be updated in real-time to Matlab.

5.4.1 OctoMap Framework

A commonly used method for 3D depth mapping is the OctoMap open-source framework [9]. Used in several robotic applications, for example for obstacle avoidance of drones [37]. This algorithm converts point cloud data into a 3D occupancy map using an octree map compression method. This method describes the volume of an environment in 3D cubes, called voxels. The volume is subdivided into eight sub-volumes, where the smallest voxels correspond with the specified resolution, see Figure 5.3. To decrease the storage space, not all root nodes are expended to leaf nodes for large areas. This method requires relatively less storage space than the point clouds, where data points from all locations are stored with the same resolution. The octree stores information about free, occupied or unknown spaces, which can later be used to determine whether or not an obstacle is located at a specific coordinate.

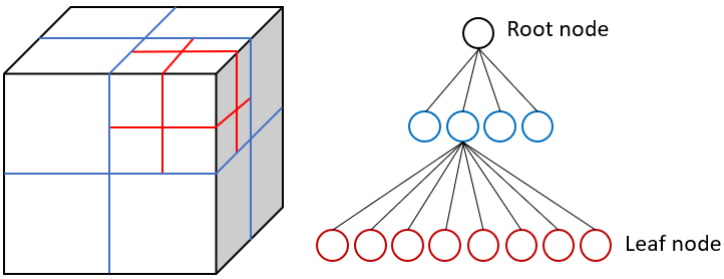


Figure 5.3: Schematic octree representation

To quantify the probability that a voxel is occupied, the log odds are used. If a voxel is seen as occupied after several scans, the log odds value of this voxel will be increased. If it exceeds a certain value, the voxel is considered occupied and will be registered in the OctoMap. This also applies the other way around when a voxel is not occupied. This probability representation of the environment reduces the effects of sensor noise on the 3D map. Modifying the specified boundaries and probability parameters, one can choose to increase the probability that a voxel is occupied. In this way, one can choose to registry static as well as dynamic objects depending on the specified boundaries and probability parameters. Given these characteristics, this OctoMap framework is further used for the use case described in Section 5.1.

5.4.2 OctoMap Implementation

To implement the OctoMap on the Gazebo setup described in Section 5.2, the ROS octo_mapping package [9] was installed on the virtual machine. With this package, a 3D occupancy grid can be generated from the data captured by the Kinect sensor. This map can be static and saved as a .bt file, or dynamic, incrementally updating the map based on the incoming point cloud data. The data from the Kinect sensor is published to the *camera/depth/points* ROS topic. This topic is added to the OctoMap launch options so that the sensor data can be used for the OctoMap server. Visualising the OctoMap generated from this server in RVis, it was noticed that the map was displayed upside down with respect to the ground plane, see Appendix B.1. This is because the data is stored relative to the camera's viewpoint, which is specified at the top of the room. To display the OctoMap in the desired orientation, a new frame must be specified that translates the viewpoint from the camera back to the origin coordinate of the Gazebo world. One way to solve this was to use the ROS tf package [34]. This package includes the command line `static_transform_publisher`, which allows transforming one frame into another frame given translations and rotations as input, shown in Appendix B.1. This node is then specified in the launch file and used for the Gazebo test setup room. Accordingly, this new coordinate frame from this ROS node is specified in the RVis as a fixed reference frame. As displayed in Figure 5.4, the OctoMap is now created with the right orientation with respect to the world frame.

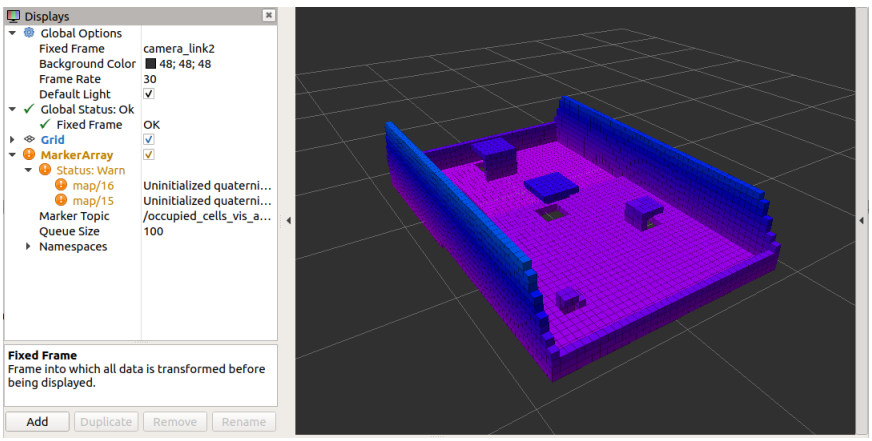


Figure 5.4: 3D OctoMap in RViz

The next step is to transfer the OctoMap data published to the ROS network on the virtual machine to Matlab, which is in the Windows environment. This is done in the same way as for the 2D grid map in Section 5.3, by connecting Matlab to the ROS network on the virtual machine. Subscribing to the */OctoMap_full* ROS topic allows receiving data published by the OctoMap server. Accordingly, this data can be used directly with the `readOccupancyMap3D` function in Matlab to create the 3D occupancy map. This map can then be plotted, as can be seen in Figure 5.5. The `checkOccupancy` command can then be used to check if the specified coordinate (in meters) is unknown (-1), obstacle-free (0) or occupied (1). The corresponding Matlab script used for this map generation can be found in Appendix B.2.

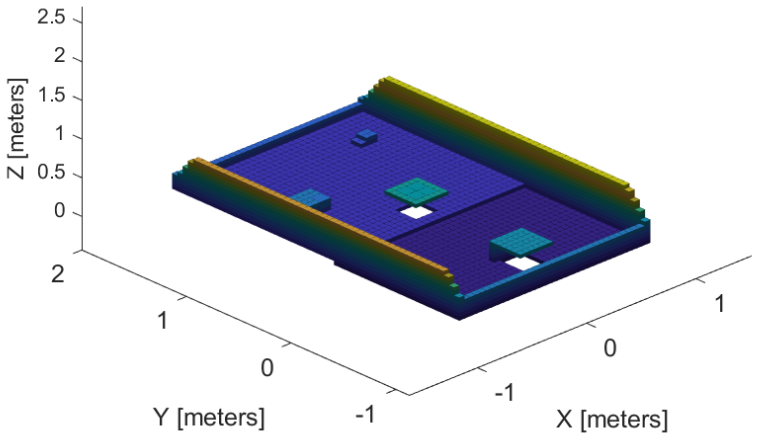


Figure 5.5: 3D Occupancy map in Matlab

5.4.3 Real Time Updating to Matlab

It was further tested how fast the 3D occupancy map can be updated in Matlab while controlling the Franka Panda Robot in the Gazebo test room, representing a dynamic obstacle. Looking at the refresh rates of the ROS topics, it was concluded that the generation of the OctoMap from the point cloud was the bottleneck. It took between 2.633 to 4.193 seconds before a new map was published to the ROS topic. This makes it difficult to capture dynamic objects on the map, especially when the objects are moving relatively fast. To increase the refresh rate of the OctoMap, the following actions were taken:

Modifying Parameters of OctoMap

As explained in Section 5.4.1, the OctoMap algorithm works with a log odds estimation, where the probability that a voxel is being occupied will be added or subtracted until the max or min from the log-space is reached. These boundaries can be changed in such a way that the sensor data will be "trusted" more, resulting in a faster updating rate of the voxels. A disadvantage of this is that any sensor noise can have a greater impact on the quality of the map. Since it is important for the use case to also register dynamic objects, the hit probability is increased and miss probability decreased in the launch file of the OctoMap. The launch file of the OctoMap including the specified parameters can be found in Appendix B.4.

The voxels that specify the ground plane in the OctoMap are not relevant for the use case and can therefore be filtered from the map in order to reduce the data points. The OctoMap plugin provides a built-in feature to filter the ground from the measured data. This filter is enabled and specified together with a reference frame in the launch file of the octoserver.

Because a detailed description of the objects in the room is not necessary for the use case, the resolution is increased from 5 to 10 centimetres. The resolution has a large impact on the refresh time of the OctoMap.

Increasing CPU cores and RAM of Virtual Machine

Because the generation of an OctoMap requires a lot of computational power, the number of CPU cores has increased from 2 to 4. In addition, the memory (RAM) is increased from 4 to 8 in the VMware Workstation settings.

Decrease Frequency of Point Clouds

Comparing the frequency from the ROS topic *camera/depth/points* (± 11 Hz) and the topic *OctoMap/-binary* (± 0.29 Hz), it was found that there is a big difference between these update rates. An extra ROS node was added to the launch file making use of the throttle function of the topic_tools ROS package [35]. This node reduces the frequency from the incoming ROS topic *camera/depth/points* from 11 Hz to 1 Hz, as specified in the launch file.

Sup-Sample Point Clouds Resolution

Another way to increase the OctoMap computation time is to sub-sample the point clouds from the Kinect sensor to a lower resolution. Since the resolution from the OctoMap is relative low (10cm), it is not necessary to have a very detailed point clouds resolution. Therefore, another ROS node is started via the launch file using the VoxelGrid filtering tool from the pcl_ros package [27]. This node receives and publishes the new point clouds to the */voxel_grid/output* topic which can then be further used for the OctoMap generation.

Combining all these actions resulted in an increase of the refresh rate from ±0.287 Hz to ±0.989 Hz of the OctoMap topic. This means that dynamic objects are better incorporated into the 3D grid map. In Appendix B.3, a ROS computation graph is shown that gives a complete overview containing all the ROS nodes and topics running during the simulation. Explaining the data transformation from the Gazebo simulation to the Matlab node in a schematic graph. This graph is made via the rqt_graph package [28]. The launch file that is used to start the nodes can be found in Appendix B.4. The final 3D occupancy map with the Panda robot and the ground filter can be seen in Figure 5.6.

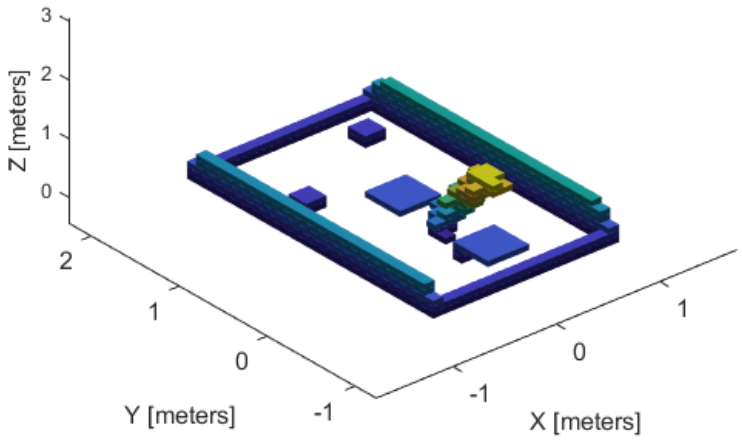


Figure 5.6: 3D Occupancy map in Matlab with Panda robot

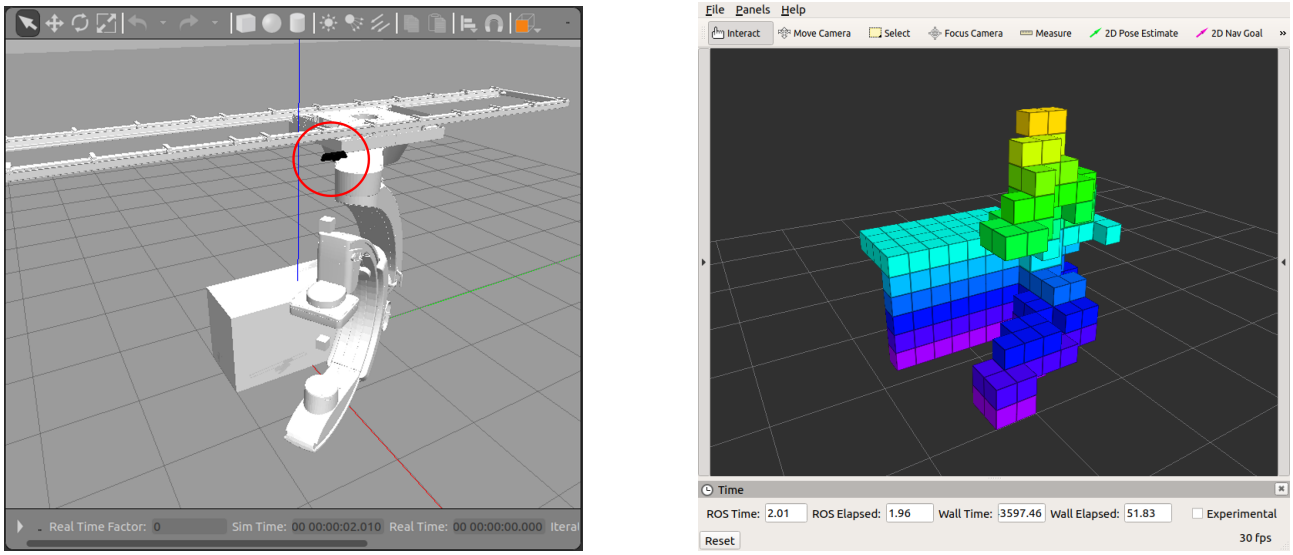
5.5 Simulation of IGT Robot

After investigating how to make a 3D grid map in Matlab from a Kinect sensor simulation in Gazebo, the perception method was further tested with the IGT robot from the use case. A new package was created on the virtual machine containing the robot files (URDF and STL files), a Gazebo world and a launch file to spawn the robot with the necessary ROS nodes. A co-simulation with Simulink was set via the Gazebo plugin in the same way as described in Chapter 3. The simulation of the robot can then be used to test the sensor configurations and to make a final recommendation. The complete simulation setup can also be used by the master student for controller validation.

5.5.1 Sensor Configurations

A Kinect sensor was added to the SDF file and attached to the top of the room next to the rails of the robot. In this way, the sensor can capture the complete top-view of the box which represents a hospital bed, see Figure 5.7a. Different sine waves for the first three joints are used to move the robot during the simulation. The OctoMap generated from the sensor data was meanwhile visualised in RViz. The refresh rate was measured around ± 1.1 seconds, making it possible to capture dynamic obstacles (the robot itself in this case). When the robot's movements were made more complex, for example by moving additional joints, the simulation speed and thus the refresh rate of OctoMap decreased. The simulation in Gazebo together with the co-simulation connection with Simulink seemed now to be the limiting factor for updating the OctoMap data. It is therefore important to ensure that the Gazebo simulation speed is not reduced too much, for example by not using a lower step size which has a large impact on the simulation speed as was demonstrated in Section 4.6. This problem will disappear if the Gazebo simulation is replaced by a real-life Kinect camera setup.

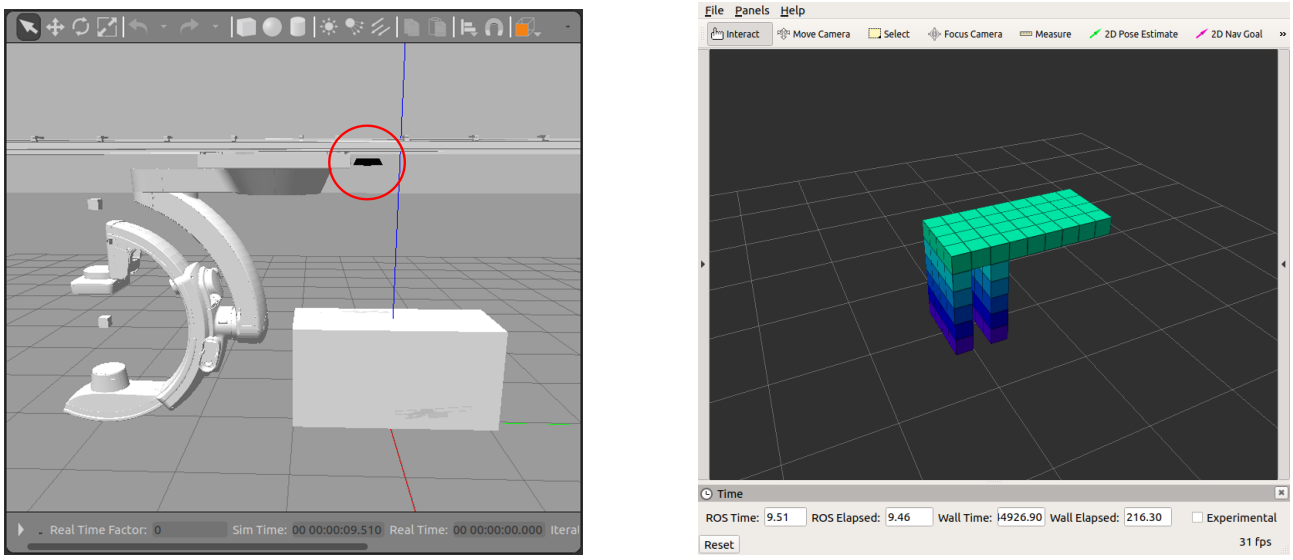
As can be seen in Figure 5.7b, a big part of the robot was sometimes visible in the OctoMap or blocking the view of the sensor. Since the robot was constantly moving, the sensor was only blocked for a small time moment. Therefore, the box was always visible in the grid map although a big part of the robot was captured as well.



(a) Gazebo simulation *(b) RViz OctoMap*

Figure 5.7: Attaching Kinect sensor to ceiling

Another sensor configuration was tested by attaching the Kinect sensor to the robot itself. It was chosen to mount the sensor on the carrier link which slides over the rails. If the camera would be attached to another link of the robot, the viewpoint of the camera would change too much, making it difficult to capture a part of the box. The robot was moved along the rails to test how the box was captured in the OctoMap, see Figure 5.8a. In Figure 5.8b, it can be seen that only a part of the object remains visible while the robot is moving. Only the visible part of the sensor will be stored in the OctoMap. This partially has to do with the custom settings described in Section 5.4.3, which ensures that the OctoMap is updated faster to also capture dynamic objects. An advantage of this configuration is that the robot is less visible, but this depends on the trajectory of the robot. It was also noticed that due to the changing camera viewpoint, the quality of the OctoMap decreased compared to the stationary camera viewpoint. As a result, sometimes voxels were shown to be occupied when they were not.



(a) Gazebo simulation *(b) RViz OctoMap*

Figure 5.8: Attaching Kinect sensor to robot

5.5.2 Filter Robot from Point Clouds

Since the position of the robot is already known during the simulation, it is not necessary to capture the geometry of the robot in the OctoMap. A solution to this would be to remove registered point clouds that represent the shape of the robot. The OctoMap can then be generated from the filtered point clouds so that only the box is visible on the map. Moveit includes an example where the UR5 robot is filtered from the point clouds captured by a Kinect sensor [24]. However, this tool is not documented in detail and it requires the ROS Noetic version. Another recently updated package is the Robot Body Filter [4]. Because it is a bit cumbersome to use, an additional Sensor Filter package [14] was found that includes a launch file to start simple nodes. This file loads the .yaml file that contains all settings and defines the ROS in and output topic to where the filtered point clouds are published to. Both packages were installed on the virtual machine and tested on the setup. However, since no comparable sensor setup was found, it was difficult to find the right settings. The example .yaml files from the plugin are applied to a fixed camera and robot frame, while for the IGT robot, it is necessary to filter the complete moving robot. Since the package is recently updated and has better documents than comparable packages, the implementation of this plugin could be further explored in future work.

6 Conclusion and Recommendation

In this study, the differences between Simscape Multibody and Gazebo have been investigated based on the comparison criteria defined in Chapter 2. The main objective was formulated as follows:

> *1. Compare Simscape Multibody and Gazebo based on defined comparison criteria by making use of co-simulation with Simulink.*

To accomplish the first main objective, comparison criteria and a robotic scenario are defined as was stated in the first sub-objective:

> *1a. Define comparison criteria and a scenario in order to compare the simulation environment based on utility, usability and performance.*

These criteria include, general characteristics, workflow, sensor possibilities, joint limits, collision modelling and simulation time. The OpenManipulator robot is chosen for the comparison since is contains elaborate documentation. Based on the implementation of this robot, it can be concluded that Gazebo requires prior knowledge about the ROS environment including the catkin workspace, while for Simscape Multibody the implementation can be done directly with an integrated function. The next sub-objective was stated as follows:

> *1b. Establish a connection between Gazebo and Simulink to perform a co-simulation with the defined scenario.*

This connection is made via the Gazebo co-simulation plugin from MathWorks. This plugin directly sends the input references from Simulink to the Gazabo topics of the robot. During the simulation, sensor information can be received in Simulink for further processing. Based on the simulation time comparison, it was shown that the simulations with the OpenManpulator take at least twice as long as real-time. This will take even longer if the simulation is extended with sensors. If simulation speed is important for a scenario, a standalone ROS node would be an alternative, as this is expected to be faster as no data transformation or synchronization between different environments is required.

All in all, the comparison gives a general idea of what is possible within the environments applied to the same scenario. It is important to note that the examples in this study only give a partial indication of what is possible in the simulation environments. The examples mainly represent general or commonly used possibilities. Based on these findings, it is recommended to use Simscape Multibody when the focus of the scenario is on the mechanical part of the robot. It provides a lot of built-in sensor blocks and additional libraries to measure and modify the dynamic properties of the robot. Built-in sensor blocks allow for directly test algorithms without needing to simulate complex virtual sensors. Sensor simulation could be further extended by, for example, integrating with Unreal Engine or Simulink 3D animation. Collisions between bodies are simulated as virtual springs, consisting of simple geometries that allow more accuracy than hard contacts. The Simulink blocks make it convenient to adjust the dynamic properties of the robot. No extra knowledge of URDF or other programming languages except Matlab and Simulink is required.

Gazebo is preferred to be used when a more complex environment needs to be simulated, such as rooms with dynamic obstacles. In addition, Gazebo supports a wide variety of sensor models that can generate synthetic data. Due to the ROS connection, algorithms can be directly tested on real hardware. By making use of the default ODE physics engine, hard contact method is used which is more robust and faster compared to soft contacts. More complex collision geometries can be simulated, such as the complete mesh of the robot but this does come at the expense of accuracy compared to soft contacts.

Apart from Gazebo, there are more external simulation environments that enable co-simulation with Simulink, for example, V-Rep, Unreal Engine and Unity. Therefore, a suggestion for future work would be to connect these environments to Simulink and compare them in terms of, for example, usability, utility and performance. This would further expand the understanding of simulation possibilities when co-simulating with Simulink.

In the second part of this study, sensor simulation in Gazebo is further explored based on a use case perspective. The second main objective of the study was as follows:

> *2. Investigate how to create a grid map of sensor simulation in Gazebo based on the requirements for the use case.*

Two ways of creating a grid map of sensor simulation in Gazebo have been investigated, in 2D and 3D. In addition, the IGT robot is further implemented in Gazebo including the Kinect sensor, so that it can be further used by the master student. An advantage of the 2D grid map is the fact that it updates fast to Matlab (around ± 0.23 seconds), making it possible to include dynamic obstacles in the map. However, since it is important to measure the distance between different body links of the robot and the objects, a 3D grid map would be preferable for the use case. The OctoMap gives the possibility to directly process the data in Matlab so that it can be further used for controller input. The refreshing time of the OctoMap has been decreased from 4.0 seconds to around 1.1 seconds. This makes it still relatively slower than the 2D grid map, but fast enough to capture moving objects in the room. For the use case, it is preferred to attach the Kinect sensor to the ceiling. Due to the relative complex movements of the robot, it is difficult to capture objects in the room without a lot of noise in the OctoMap.

Finding the best sensor configuration requires more research. Since the IGT robot is relatively large, it is difficult to capture all objects in a room without blocking the sensor's view. For future work, the 3D representation of the room could be more detailed by combining point clouds from multiple Kinect sensors. This would also make the 3D map more detailed, allowing to capture objects from multiple viewpoint angles. Another suggestion for future work is to filter the robot's geometry from the map. Since the position of the robot is already known, it is not necessary to capture the robot in the OctoMap. Suggestions for ROS packages can be found in Section 5.5.2.

Bibliography

[1] Ahmed R.J. Almusawi, L. Canan Dülger and Sadettin Kapucu. "Robotic arm dynamic and simulation with Virtual Reality Model (VRM)". In: *International Conference on Control, Decision and Information Technologies, CoDIT 2016* (Oct. 2016), pp. 335–340. DOI: 10.1109/CODIT.2016.7593584.

[2] Heesun Choi et al. "On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward". In: *Perspective* 118.1 (Sept. 2020). DOI: 10.1073/pnas.1907856118/-/DCSupplemental.

[3] Mirella Santos Pessoa De Melo et al. "Analysis and comparison of robotics 3D simulators". In: *Proceedings - 2019 21st Symposium on Virtual and Augmented Reality, SVR 2019*. Institute of Electrical and Electronics Engineers Inc., Oct. 2019, pp. 242–251. ISBN: 9781728154343. DOI: 10.1109/SVR.2019.00049.

[4] Tomas Petricek Eitan Marder-Eppstein. *robot_body_filter - ROS Wiki*. URL: http://wiki.ros.org/robot_body_filter.

[5] Gazebo API Reference. *Gazebo: Sensors*. URL: https://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/group__gazebo__sensors.html.

[6] *gazebo_ros_pkgs - ROS Wiki*. URL: http://wiki.ros.org/gazebo_ros_pkgs.

[7] *GitHub - ROBOTIS-GIT/open_manipulator: OpenManipulator for controlling in Gazebo and Moveit with ROS*. URL: https://github.com/ROBOTIS-GIT/open_manipulator.

[8] Victor I C Hofstede, Bachelor Opleiding and Kunstmatige Intelligentie. *The importance and purpose of simulation in robotics*. Tech. rep. Amsterdam: University of Amsterdam, June 2015.

[9] Armin Hornung et al. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous Robots* 34.3 (Feb. 2013), pp. 189–206. ISSN: 09295593. DOI: 10.1007/S10514-012-9321-0.

[10] Hyejong Kim. *open_manipulator - ROS Wiki*. URL: http://wiki.ros.org/open_manipulator.

[11] Ehsan Izadi and Adam Bezuijen. "Simulating direct shear tests with the Bullet physics library: A validation study". In: *PLoS ONE* 13.4 (Apr. 2018). ISSN: 19326203. DOI: 10.1371/journal.pone.0195073.

[12] Nathan Koenig and Andrew Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* 3 (2004), pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727.

[13] Jizhan Liu et al. "Experiments and analysis of close-shot identification of on-branch citrus fruit with realsense". In: *Sensors* 18.5 (May 2018). ISSN: 14248220. DOI: 10.3390/s18051510.

[14] Martin Pecka. *sensor_filters - ROS Wiki*. URL: http://wiki.ros.org/sensor_filters.

[15] MathWorks. *Connect a Physical Model in Simulink to Unreal Engine | AUV Deep Dive, Part 6 - YouTube*. May 2021. URL: https://www.youtube.com/watch?v=fNdOfVYxkGg.

[16] MathWorks. *Simscape Multibody - MATLAB & Simulink*. URL: https://nl.mathworks.com/products/simscape-multibody.html.

[17] MathWorks Student Competitions Team. *Designing Robot Manipulator Algorithms - File Exchange - MATLAB Central*. Oct. 2019. URL: https://nl.mathworks.com/matlabcentral/fileexchange/65316-designing-robot-manipulator-algorithms.

[18] MathWorks Support. *Gazebo Simulation for Robotics System Toolbox - MATLAB & Simulink - MathWorks Benelux*. URL: https://nl.mathworks.com/help/robotics/ug/gazebo-simulation-for-robotics-system.html.

[19] MathWorks Support. *How Gazebo Simulation for Robotics System Toolbox Works - MATLAB & Simulink - MathWorks Benelux*. URL: https://nl.mathworks.com/help/robotics/ug/how-gazebo-simulation-for-robotics-works.html.

[20] MathWorks Support. *Perform Co-Simulation between Simulink and Gazebo - MATLAB & Simulink - MathWorks Benelux*. URL: https://nl.mathworks.com/help/robotics/ug/perform-co-simulation-between-simulink-and-gazebo.html.

[21] MathWorks Support. *ROS 2 Dashing and Gazebo - MATLAB & Simulink*. URL: https://nl.mathworks.com/support/product/robotics/ros2-vm-installation-instructions-v5.html.

[22] MathWorks Support. *URDF Import - MATLAB & Simulink - MathWorks Benelux*. URL: https://nl.mathworks.com/help/physmod/sm/ug/urdf-import.html#bvmwhdm-1.

[23] Mathworks Support. *Modeling Contact Force Between Two Solids - MATLAB & Simulink - MathWorks Benelux*. URL: https://nl.mathworks.com/help/physmod/sm/ug/modeling-contact-force-between-two-solids.html.

[24] MoveIt. *Mesh Filter with UR5 and Kinect — moveit_tutorials Noetic documentation*. URL: https://ros-planning.github.io/moveit_tutorials/doc/mesh_filter/mesh_filter_tutorial.html.

[25] Dragomir N. Nenchev, Atsushi Konno and Teppei Tsujita. "Simulation". In: *Humanoid Robots*. Elsevier, 2019, pp. 421–471. ISBN: 9780128045602. DOI: 10.1016/B978-0-12-804560-2.00015-8. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780128045602000158.

[26] *OpenMANIPULATOR-X*. URL: https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/.

[27] *pcl_ros - ROS Wiki*. URL: https://wiki.ros.org/pcl_ros.

[28] *rqt_common_plugins - ROS Wiki*. URL: https://wiki.ros.org/rqt_common_plugins.

[29] U Schmucker et al. *Contact processing in the simulation of the multi-body systems*. Tech. rep. Magdeburg, Germany: University of Magdeburg, Institute for Electrical Energy Systems, Dec. 2008. URL: https://www.researchgate.net/publication/228699941.

[30] Sebastian Castro. *Walking Robot Modeling and Simulation » Student Lounge - MATLAB & Simulink*. Dec. 2019. URL: https://blogs.mathworks.com/student-lounge/2019/12/20/walking-robot-modeling-and-simulation/.

[31] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System, ROS Melodic Morenia*. May 2018. URL: https://www.ros.org/.

[32] Aaron Staranowicz and Gian Luca Mariottini. "A Survey and Comparison of Commercial and Open-Source Robotic Simulator Software". In: *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments - PETRA '11* (May 2011). DOI: 10.1145/2141622.

[33] Kenta Takaya et al. "Simulation environment for mobile robots testing using ROS and Gazebo". In: *2016 20th International Conference on System Theory, Control and Computing, ICSTCC 2016 - Joint Conference of SINTES 20, SACCS 16, SIMSIS 20 - Proceedings* (Dec. 2016), pp. 96–101. DOI: 10.1109/ICSTCC.2016.7790647.

[34] *tf - ROS Wiki*. URL: http://wiki.ros.org/tf.

[35] *topic_tools - ROS Wiki*. URL: http://wiki.ros.org/topic_tools.

[36] M. Torres-Torriti, T. Arredondo and P. Castillo-Pizarro. "Survey and comparative study of free simulation software for mobile robots". In: *Robotica* 34.4 (Apr. 2016), pp. 791–822. ISSN: 14698668. DOI: 10.1017/S0263574714001866.

[37] Simon Vanneste, Ben Bellekens and Maarten Weyn. *3DVFH+: Real-Time Three-Dimensional Obstacle Avoidance Using an Octomap*. Tech. rep. Antweropen: CoSys-Lab, Faculty of Applied Engineering, 2014.

[38] Andres Vivas and Jose Maria Sabater. "UR5 Robot Manipulation using Matlab/Simulink and ROS". In: *2021 IEEE International Conference on Mechatronics and Automation, ICMA 2021* (Aug. 2021), pp. 338–343. DOI: 10.1109/ICMA52036.2021.9512650.

[39] Maida Cohodar Nedzma Kobilica Vjekoslav Damic. "Development of Dynamic Model of Robot with Parallel Structure Based on 3D CAD Model, Proceedings of the 30th DAAAM International Symposium". In: *Published by DAAAM International* (2019), pp. 155–0160. ISSN: 1726-9679. DOI: 10.2507/30th.daaam.proceedings.020.

[40] Guido Wolfs. *Co-Simluation Simulink and Gazebo Examples · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Co-Simluation%20Simulink%20and%20Gazebo%20Examples.

[41] Guido Wolfs. *Gazebo Perception Files · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Gazebo%20Perception%20Files.

[42] Guido Wolfs. *Scenario OpenManipulator Files · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Scenario%20OpenManipulator%20Files.

[43] Guido Wolfs. *Simulink Models · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Simulink%20Models.

[44] Guido Wolfs. *Simulink Models/Collision Comparison · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Simulink%20Models/Collision%20Comparison.

[45] Guido Wolfs. *Simulink Models/Joint-limit Comparison · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Simulink%20Models/Joint-limit%20Comparison.

[46] Guido Wolfs. *Simulink Models/Sensor Comparison · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Simulink%20Models/Sensor%20Comparison.

[47] Guido Wolfs. *Simulink Models/Time Comparison · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Simulink%20Models/Time%20Comparison.

[48] Guido Wolfs. *Virtual Machine · main · ETProjects / GW CompSimEnv · GitLab*. 2022. URL: https://gitlab.tue.nl/et_projects/gw-compsimenv/-/tree/main/Virtual%20Machine.

A Co-Simulation and Comparison

This appendix contains information about ROS, the virtual machine and the Simulink main model that forms the basis for the comparison. Detailed information on the computer setup used for the time measurements is also provided.

A.1 Robot Operating System

The Robot Operating System (ROS) is an open-source framework for building robotic systems. It includes a huge repository with algorithms, packages and drivers. ROS ensures that different robotic components, like actuators, sensors and controllers are connected to each other and can exchange information. A simple ROS setup can be seen in Figure A.1. Each ROS network consists of a ROS master, which allows for communication between the different processes. This information goes via ROS topics. In this way, node 2 can receive information that is published by node 1. It is also possible to have nodes that both publish and receive information. More information about ROS can be found at [31].

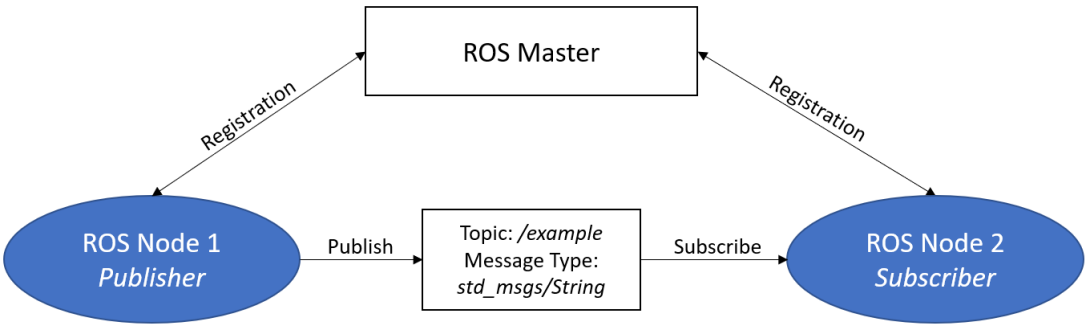


Figure A.1: Basic schematic ROS setup

A.2 Virtual Machine

A virtual machine is a computer program that runs as a "guest" on a host computer. This makes it possible to use other operating systems and programmes that do not have to be installed on the main computer. The virtual machine is uploaded to the repository [48]. The following main elements are included in the machine:

- Ubuntu 18.04.6 LTS.
- ROS distribution Melodic.
- ROS 2 Dashing.
- Gazebo 9.19.0-1 bionic.
- Example Gazebo worlds from MathWorks.
- World files and robot models used for manipulator scenario.
- World files and packages used for the perception in Gazebo.

A.3 Co-Simulation Setup Details

Having a connection between the OpenManipulator in Gazebo and Simulink, it is possible to send data using Matlab commands or by using Simulink blocks. Different types of data to Gazebo such as position, velocity and torque can be sent as a reference signal to the robot. Via the co-simulation functions from Matlab, it is possible to manipulate the pose of the robot using the following function:

```
[status,message] = gzjoint('set','robot','joint1','Axis','0','Angle',pi/4)
```

This function directly changes the position of the first joint of the manipulator from 0 to pi/4. For this command, neither a connection with Simulink nor a feedback controller is needed. Other types of commands can be found in the the MathWorks support documentation, see [18].

A.3.1 Simulink Sending Data to Gazebo

To send the input reference signal from Simulink to Gazebo, several blocks are needed. These are required to control each joint of the OpenManipulator separately. In Figure A.2, the Simulink model can be seen that sends the reference input to the robot in Gazebo.

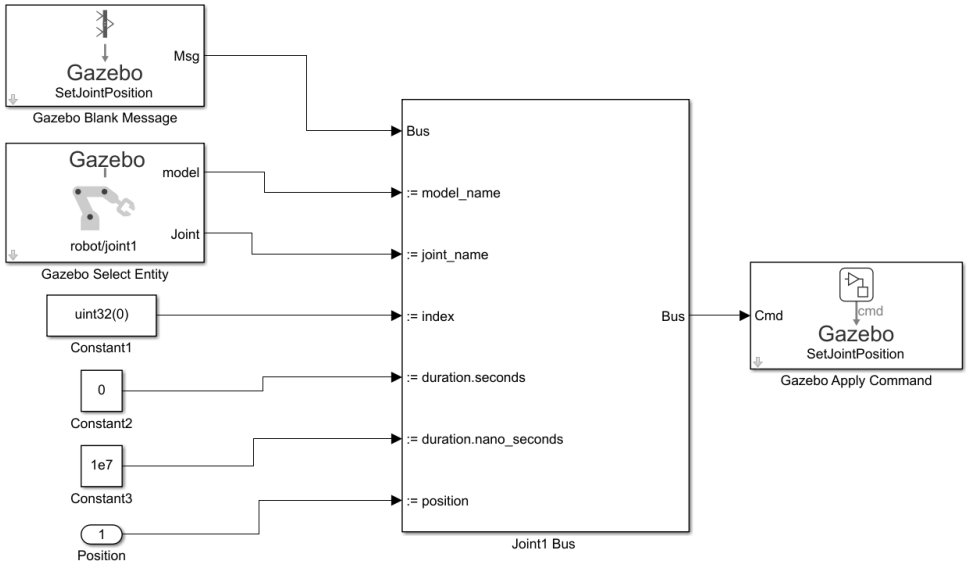


Figure A.2: Set joint position in Simulink

The Gazebo Apply Command block on the right sends the data to the plugin in the Linux environment. This block receives data collected by the bus assignment, which consists of the following elements:

- **Gazebo Blank Message block**: Creates a blank Gazebo message or command. For the manipulator scenario, "SetJointPostion", "SetJointVelocity", "ApplyJointTorque" are used since these are needed to control the revolute and prismatic joints.
- **Gazebo Select Entity bock**: A joint or link from the Gazebo model can be selected to be actuated. These topics are Gazebo topics and can be directly found in Simulink. There is no ROS node required to transfer the data, this is all done by the plugin itself. The type of topics depends on the possibilities of the robot in the Gazebo environment. For the OpenManipulator the following options can be chosen, see Figure A.3. Only the joints 1 up to 4 and the gripper topics are used for the scenario.

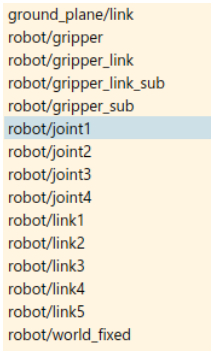


Figure A.3: OpenManipulator Gazebo topics

- **Constant block with "uint32(0)"**: Indicates which axes will be actuated. Since the OpenManipulator has only 1 degree of freedom joints, this value will be set to "uint32(0)". This indicates that the first axes are actuated.
- **Constant block with "0" and "1e7" duration**: With these blocks, the duration of the applied input (in this case position) is defined. Two separate blocks are used to increase precision. For the OpenManipulator, the duration of the input signal is chosen to be equal to the sample time which is specified as "0.01" seconds. Therefore the duration was set to 1e7 nanoseconds. When increasing the sample time of the mode, also the duration of the reference input can be increased accordingly as was done for the simulation time comparison in Section 4.6.
- **Position**: The reference position data that the joints need to follow. This element specifies in this case the chosen position of the OpenManipulator. This port can also be changed to velocity and torque if another type of reference input is required. For example, when changing the type of reference from position to torque, it is also required to change the Gazebo message block.

A.3.2 Simulink Receiving Data from Gazebo

Receiving data from a chosen Gazebo topic goes via a Gazebo Read block. Position and velocity data from the joints are extracted using a bus selector. With this block, also data from sensors located in the Gazebo environment can be received and used for the sensor comparison in Section 4.4.

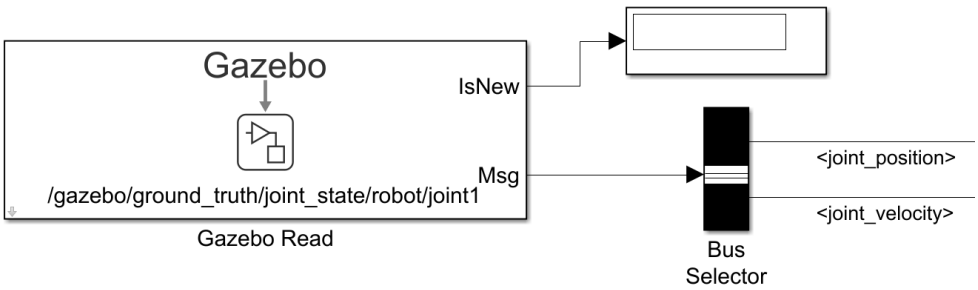


Figure A.4: Receiving data in Simulink from Gazebo

A.4 Main Simulink Model

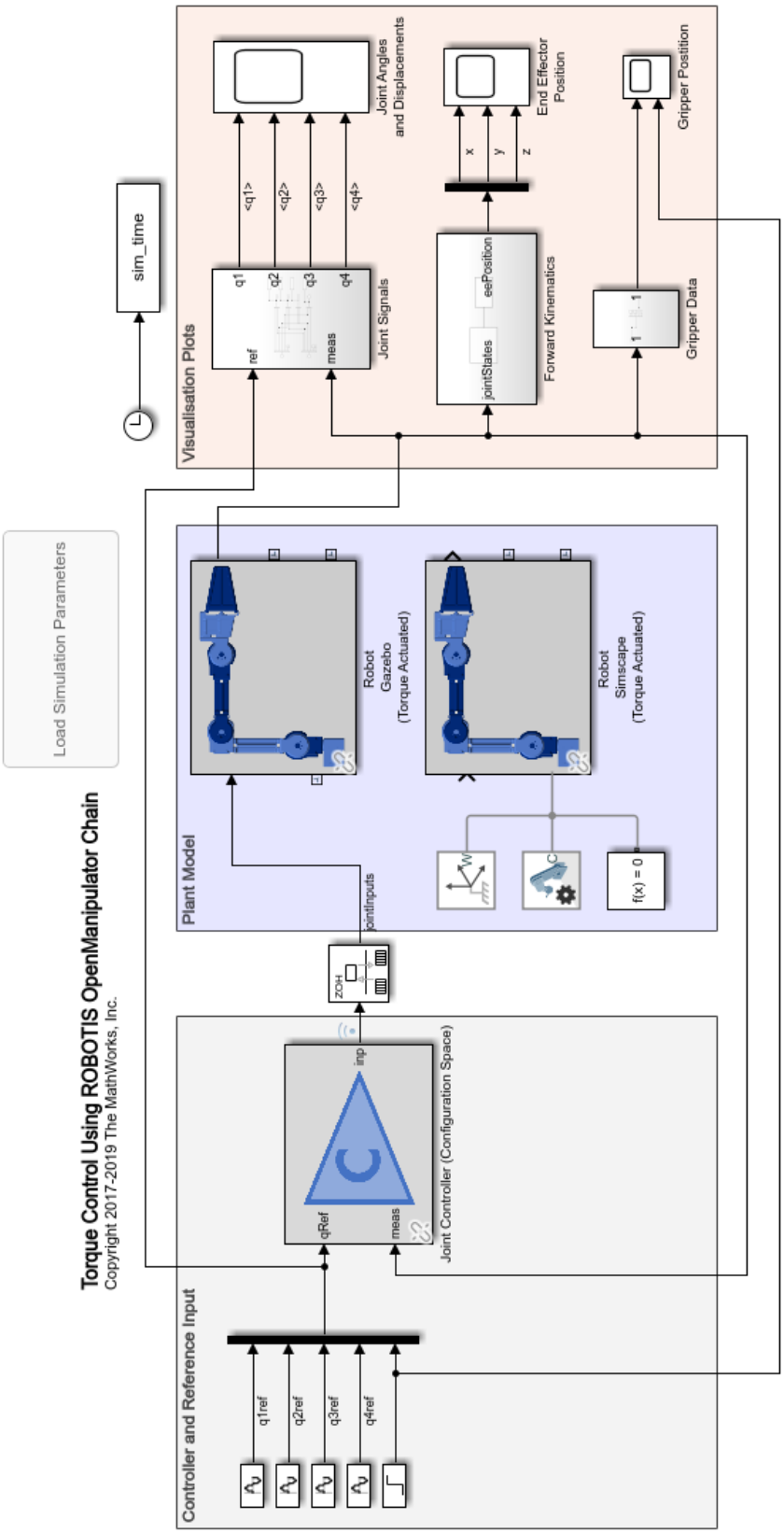


Figure A.5: Simulink model with two different plants [17]

A.5 Time Measurement Configuration

The time measurement setup consists of a dual monitor setup, displaying Simulink on the first screen and the robot on the second screen. Only Simulink and the virtual machine are running during the measurements. The virtual machine is only used for the Gazebo measurements.

Table A.1: PC specification

OS	Windows 10 64bit Edition
CPU	Intel Core i5 8600K
GPU	ASUS GeForce GTX 1070 Ti
Memory	HyperX Predator - DDR4 - 16GB
SSD	Samsung 970 EVO Plus M.2 - SSD - 1TB

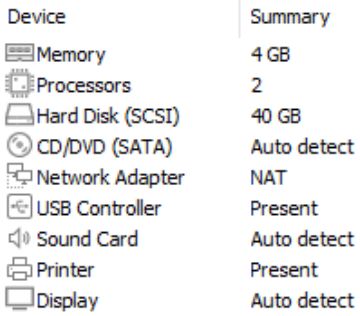


Figure A.6: VMware workstation settings

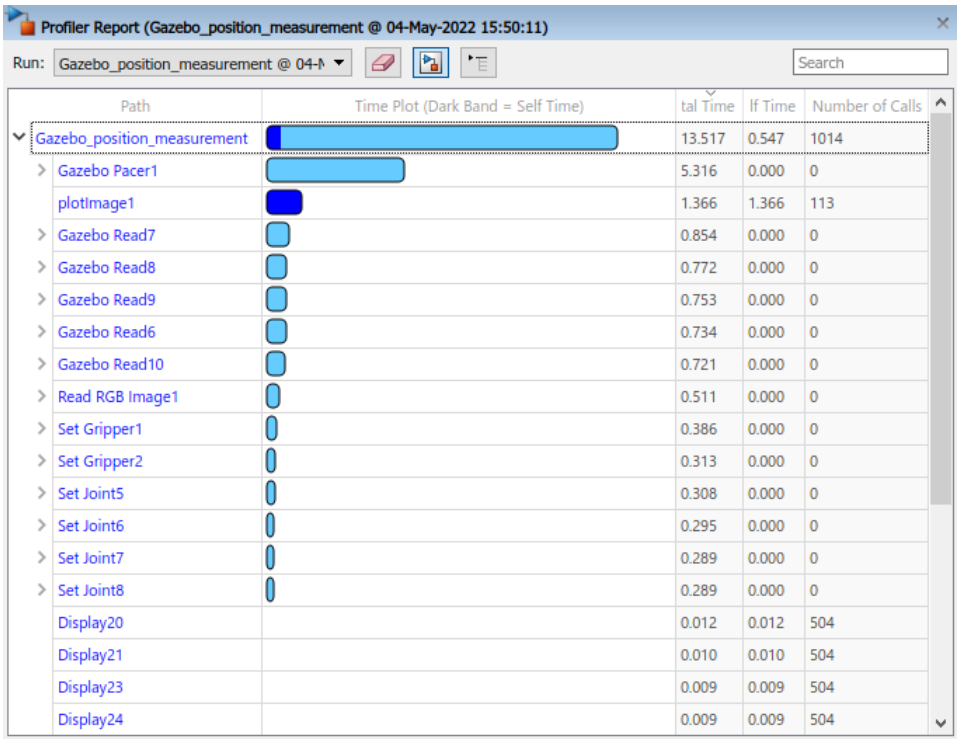


Figure A.7: Time measurement example with Simulink Profiler

B Gazebo Perception

In this appendix, RViz visualisations are included to demonstrate the coordinate transformation from the sensor frame. In addition, Matlab codes for the grid maps, a ROS computation graph and the launch file that is created for the sensor simulation in Gazebo are included.

B.1 RViz Visualisations

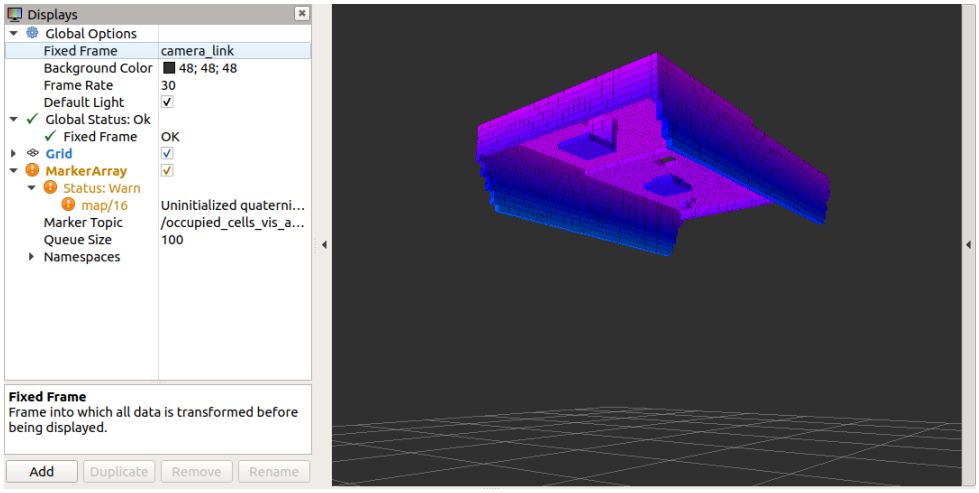


Figure B.1: 3D Octomap in RViz before coordinate transformation

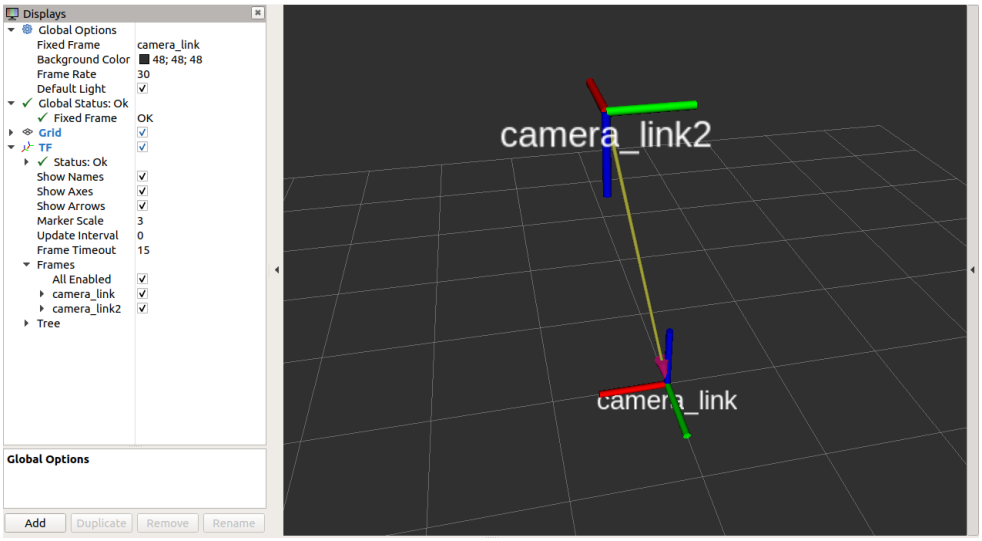


Figure B.2: RViz coordinate transformation camera_link2

B.2 Matlab Scripts for 2D/3D Grid Map

Listing B.1: MATLAB script - 2D grid map

```
clear all; close all; clc;
%% Make a connection to ROS server and show ROS topics
rosinit('192.168.21.134')
rostopic list

%% Reading PNG image
imageData = imread('top_view_grey.png');
imageData(imageData<250) = 0;
imshow(imageData)

%% Convert to grayimage
grayimage = rgb2gray(imageData);
bwimage = grayimage < 0.5;

%% Create and plot grid map
% Set the resolution, this value is calculated with spatial_calibration_demo.m file
resolution = 245.182076;

grid = binaryOccupancyMap(bwimage,resolution);
inflate(grid, 0.05)

show(grid)

%% Checking for obstacle
occVal = checkOccupancy(grid,[1.2 1])
```

Listing B.2: MATLAB script - 3D grid map

```
clear all; close all; clc;
%% Make a connection to ROS server and show ROS topics
rosinit('192.168.21.134')
rostopic list
rosshutdown

%% Subscribe to ROS topic and make OccupancyMap
map_topic_info_full = rossubscriber('/octomap_full');
map_topic_info_binary = rossubscriber('/octomap_binary');
map_message = receive(map_topic_info_binary);
map = readOccupancyMap3D(map_message);
% r_inflation = 0.05
% inflate(map, r_inflation);

show(map);
save 2D_gridmap map

%% Loading mapFile
load('mapFile_translated.mat')
grid on;
show(map)

%% Check if location is occupied
% Occupancy values can be obstacle-free (0), occupied (1), or unknown (1).
% Each row of the array XYZ corresponds to a point with [X Y Z] world coordinates.

% Check if the coordinate is occupied
OccVal1 = checkOccupancy(map,[0 0 0])

% Check probabliltiy of location being occupied
OccVal1 = getOccupancy(map,[0 0 0])
```

B.3 ROS Computation Graph

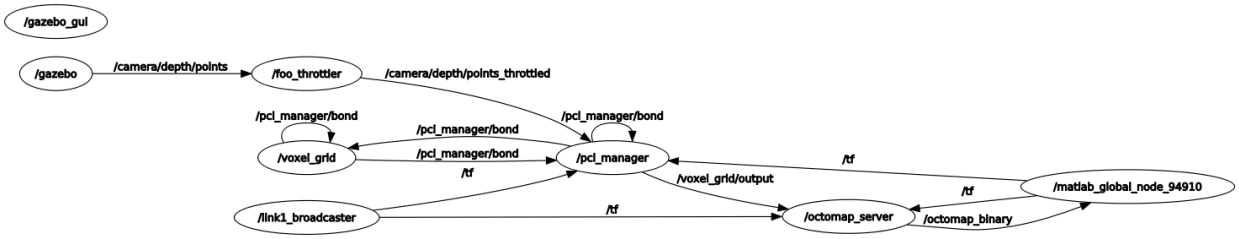


Figure B.3: ROS qrt computation graph (nodes only) [28]

B.4 Launch Files

Listing B.3: XML code - Launch file for RGBd_room

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
    <!-- Set standard world -->
    <arg name="world" default="empty"/>

    <!-- Starting rviz -->
    <node type="rviz" name="rviz" pkg="rviz" args="-d rviz_files/rviz/depth_camera.rviz" />

    <!-- Selecting world -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
      <arg name="world_name"  value="$(find guido)/worlds_usecase/$(arg world).world" />
    </include>

    <!-- Change ref frame -->
    <node pkg="tf" type="static_transform_publisher" name="link1_broadcaster" args="0.5 0 2.5 -1.57079
     0 3.14159 camera_link camera_link2 100" />

    <!-- Start throttle point clouds to 1 Hz -->
    <node name="foo_throttler" type="throttle" pkg="topic_tools" args="messages /camera/depth/points 1
     /camera/depth/points_throttled" />

    <!-- Downsample Pointclouds -->
    <node pkg="nodelet" type="nodelet" name="pcl_manager" args="manager" output="screen" />

    <!-- Run a VoxelGrid filter to clean NaNs and downsample the data -->
    <node pkg="nodelet" type="nodelet" name="voxel_grid" args="load pcl/VoxelGrid pcl_manager" output=
    "screen">
    <remap from="~input" to="/camera/depth/points_throttled" />
       <rosparam>
          filter_field_name: x
          filter_limit_min: -3
          filter_limit_max: 3
          filter_limit_negative: False
          leaf_size: 0.05
       </rosparam>
    </node>

  <!-- Find Panda robot -->
  <param name ="robot_description" command= "cat '$(find franka_panda_description)/robots/panda_arm.
    urdf'" />

  <!-- Import Panda robot -->
  <node
```

```
    name="spawn_model"
    pkg="gazebo_ros"
    type="spawn_model"
    args="-urdf -param robot_description -model panda_arm"
    output="screen" />

</launch>
```

Listing B.4: XML code - Launch file for OctoMap including specified parameters

```
<launch>
  <node pkg="octomap_server" type="octomap_server_node" name="octomap_server">
    <param name="resolution" value="0.1" />

    <!-- fixed map frame (set to 'map' if SLAM or localization running!) -->
    <param name="frame_id" type="string" value="camera_link2" />
    <param name="base_frame_id" type="string" value="camera_link2" />

    <!-- maximum range to integrate (speedup!) -->
    <param name="sensor_model/max_range" value="5.0" />

    <!-- data source to integrate (PointCloud2) -->
    <remap from="cloud_in" to="/voxel_grid/output" />

    <!-- Set parameters -->
    <param name="sensor_model/hit"  value="1" />
    <param name="sensor_model/miss"  value="0.01" />
    <param name="sensor_model/min"  value="0.49" />
    <param name="sensor_model/max"  value="0.5" />

    <param name="filter_ground" type="bool" value="true" />
  </node>
</launch>
```

C Gazebo Models

In this appendix, the XML codes that are used for the Gazebo models are included. These codes can be added directly to an SDF file for a Gazebo simulation. The complete SDF files containing these sensor models and the URDF for the OpenManipulator can be found in the repository [42].

C.1 Plugin used for SDF files

Listing C.1: XML code - Gazebo plugin

```
<plugin name='GazeboPlugin' filename='lib/libGazeboCoSimPlugin.so'>
     <portNumber>14581</portNumber>
   </plugin>
```

C.2 Lidar Sensor Model

Listing C.2: XML code - Lidar sensor

```
<!-- Lidar Sensor Model -->
  <model name="hokuyo0">
    <link name="link">
     <pose>0.25 0.25 0.05 0 0 0</pose>
      <gravity>false</gravity>
      <inertial>
        <mass>0.1</mass>
      </inertial>
      <visual name="visual">
        <geometry>
          <mesh>
            <uri>model://hokuyo/meshes/hokuyo.dae</uri>
          </mesh>
        </geometry>
      </visual>
      <sensor name="laser" type="ray">
        <pose>0.01 0 0.03 0 -0 0</pose>
        <ray>
          <scan>
            <horizontal>
              <samples>640</samples>
              <resolution>1</resolution>
              <min_angle>-3.14</min_angle>
              <max_angle>3.14</max_angle>
            </horizontal>
          </scan>
          <range>
            <min>0.08</min>
            <max>10</max>
            <resolution>0.01</resolution>
          </range>
          <noise>
            <type>gaussian</type>
            <mean>0.0</mean>
            <stddev>0.01</stddev>
          </noise>
        </ray>
        <always_on>1</always_on>
        <update_rate>200</update_rate>
```

```
        <visualize>true</visualize>
      </sensor>
    </link>
  </model>-
```

C.3 IMU Sensor Model

Listing C.3: XML code - IMU sensor

```
<!-- IMU Sensor Model -->
    <link name="imu_sensor_link">
     <pose>0.22 0 0.25 0 -0 0</pose>
      <inertial>
        <mass>0.0001</mass>
      </inertial>
      <visual name="visual">
        <geometry>
          <box>
            <size>0.01 0.01 0.01</size>
          </box>
        </geometry>
      </visual>
      <collision name="collision">
        <geometry>
          <box>
            <size>0.01 0.01 0.01</size>
          </box>
        </geometry>
      </collision>

      <sensor name="imu" type="imu">
        <imu>
          <angular_velocity>
            <x>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>2e-4</stddev>
                <bias_mean>0.0000075</bias_mean>
                <bias_stddev>0.0000008</bias_stddev>
              </noise>
            </x>
            <y>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>2e-4</stddev>
                <bias_mean>0.0000075</bias_mean>
                <bias_stddev>0.0000008</bias_stddev>
              </noise>
            </y>
            <z>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>2e-4</stddev>
                <bias_mean>0.0000075</bias_mean>
                <bias_stddev>0.0000008</bias_stddev>
              </noise>
            </z>
          </angular_velocity>
          <linear_acceleration>
            <x>
              <noise type="gaussian">
```

```
                <mean>0.0</mean>
                <stddev>1.7e-2</stddev>
                <bias_mean>0.1</bias_mean>
                <bias_stddev>0.001</bias_stddev>
              </noise>
            </x>
            <y>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>1.7e-2</stddev>
                <bias_mean>0.1</bias_mean>
                <bias_stddev>0.001</bias_stddev>
              </noise>
            </y>
            <z>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>1.7e-2</stddev>
                <bias_mean>0.1</bias_mean>
                <bias_stddev>0.001</bias_stddev>
              </noise>
            </z>
          </linear_acceleration>
        </imu>
        <always_on>1</always_on>
        <update_rate>200</update_rate>
      </sensor>
    </link>

<!-- Sensor joint -->
      <joint name='imu_sensor_joint' type='fixed'>
    <pose>0.16 0 0.2045 0 -0 0</pose>
    <parent>link5</parent>
        <child>imu_sensor_link</child>
      </joint>
```

C.4 RGB Camera Model

Listing C.4: XML code - camera sensor

```
<!-- Camera Sensor Model -->
   <model name="camera0">
    <pose>0 1 0.1 0 0 -1.57079633</pose>
    <link name="link">
      <inertial>
        <mass>0.1</mass>
        <inertia>
          <ixx>0.000166667</ixx>
          <iyy>0.000166667</iyy>
          <izz>0.000166667</izz>
        </inertia>
      </inertial>
      <collision name="collision">
        <geometry>
          <box>
            <size>0.1 0.1 0.1</size>
          </box>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
```

```
22          <box>
23            <size>0.1 0.1 0.1</size>
24          </box>
25        </geometry>
26      </visual>
27      <sensor name="camera" type="camera">
28        <camera>
29          <horizontal_fov>1.047</horizontal_fov>
30          <image>
31            <width>320</width>
32            <height>240</height>
33          </image>
34          <clip>
35            <near>0.1</near>
36            <far>100</far>
37          </clip>
38        </camera>
39        <always_on>1</always_on>
40        <update_rate>200</update_rate>
41        <visualize>true</visualize>
42      </sensor>
43    </link>
44  </model>
```

C.5 Collision Settings

Listing C.5: XML code - collision settings in Gazebo

```
1       <collision name='link1_collision'>
2           <pose frame=''>0 0 0 0 -0 0</pose>
3           <geometry>
4             <mesh>
5               <scale>0.001 0.001 0.001</scale>
6               <uri>/home/user/catkin_ws/src/open_manipulator/open_manipulator_description/meshes/
    chain_link1.stl</uri>
7             </mesh>
8           </geometry>
9           <surface>
10            <contact>
11              <ode>
12                <kp>1e+06</kp>
13                <kd>100</kd>
14                <max_vel>1</max_vel>
15                <min_depth>0.001</min_depth>
16              </ode>
17            </contact>
18            <friction>
19              <ode>
20                <mu>30</mu>
21                <mu2>30</mu2>
22              </ode>
23              <torsional>
24                <ode/>
25              </torsional>
26            </friction>
27            <bounce/>
28          </surface>
29          <max_contacts>10</max_contacts>
30        </collision>
```

C.6 Kinect Sensor Model

Listing C.6: XML code - Kinect sensor with fixed joint attachment

```
<!-- Kinect Sensor Model -->
   <model name='depth_camera'>
      <pose frame=''>0.5 0 2.5 -1.57079 1.57079 3.14159</pose>
      <link name='camera_sensor_link'>
        <inertial>
          <mass>0.1</mass>
          <inertia>
            <ixx>0.000166667</ixx>
            <iyy>0.000166667</iyy>
            <izz>0.000166667</izz>
            <ixy>0</ixy>
            <ixz>0</ixz>
            <iyz>0</iyz>
          </inertia>
          <pose frame=''>0 0 0 0 -0 0</pose>
        </inertial>
        <collision name='collision'>
          <geometry>
            <box>
              <size>0.073 0.276 0.072</size>
            </box>
          </geometry>
          <max_contacts>10</max_contacts>
          <surface>
            <contact>
              <ode/>
            </contact>
            <bounce/>
            <friction>
              <torsional>
                <ode/>
              </torsional>
              <ode/>
            </friction>
          </surface>
        </collision>
        <visual name='visual'>
          <geometry>
            <mesh>
              <uri>model://kinect/meshes/kinect.dae</uri>
            </mesh>
          </geometry>
        </visual>
        <sensor name='RGB_camera1' type='depth'>
          <update_rate>20</update_rate>
          <camera name='__default__'>
            <horizontal_fov>1.0472</horizontal_fov>
            <image>
              <width>640</width>
              <height>480</height>
              <format>R8G8B8</format>
            </image>
            <clip>
              <near>0.05</near>
              <far>3</far>
            </clip>
          </camera>
          <plugin name='camera_plugin' filename='libgazebo_ros_openni_kinect.so'>
            <baseline>0.2</baseline>
```

```
60            <alwaysOn>1</alwaysOn>
61            <updateRate>0.0</updateRate>
62            <cameraName>camera_ir</cameraName>
63            <imageTopicName>/camera/color/image_raw</imageTopicName>
64            <cameraInfoTopicName>/camera/color/camera_info</cameraInfoTopicName>
65            <depthImageTopicName>/camera/depth/image_raw</depthImageTopicName>
66            <depthImageCameraInfoTopicName>/camera/depth/camera_info</depthImageCameraInfoTopicName>
67            <pointCloudTopicName>/camera/depth/points</pointCloudTopicName>
68            <frameName>camera_link</frameName>
69            <pointCloudCutoff>0.5</pointCloudCutoff>
70            <pointCloudCutoffMax>3.0</pointCloudCutoffMax>
71            <distortionK1>0</distortionK1>
72            <distortionK2>0</distortionK2>
73            <distortionK3>0</distortionK3>
74            <distortionT1>0</distortionT1>
75            <distortionT2>0</distortionT2>
76            <CxPrime>0</CxPrime>
77            <Cx>0</Cx>
78            <Cy>0</Cy>
79            <focalLength>0</focalLength>
80            <hackBaseline>0</hackBaseline>
81          </plugin>
82        </sensor>
83        <self_collide>0</self_collide>
84        <enable_wind>0</enable_wind>
85        <kinematic>0</kinematic>
86      </link>
87      <joint name='RGBd_joint' type='fixed'>
88        <pose frame=''>2 2 2 0 -0 0</pose>
89        <parent>world</parent>
90        <child>camera_sensor_link</child>
91      </joint>
92    </model>
```