ENG3163: Car-in-the-loop simulation by implementation of ROS2 and Gazebo

UNIVERSITY OF SURREY

Faculty of Engineering & Physical Sciences
Department of Mechanical Engineering Sciences
Undergraduate Programmes in Mechanical Engineering Sciences

Document:	Undergraduate Final Year Project
Name:	Shea Tanner-Cowie
URN:	6596634
Project leader:	Dr Saber Fallah

Personal Statement of Originality: I confirm that the submitted work is my own work. No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have also clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook. I agree that the University may submit my work to means of checking this, such as the plagiarism detection service Turnitin® UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Shea Tanner-Cowie.

Abstract

Recent developments in the field of robotics have led to a new requirement to test robotic systems more efficiently. Therefore, this document aimed to provide research around a method of robotic testing and reported on a built example, using this method on one of the faculty's robots. The method used applications called ROS2 and Gazebo to make an accurate, virtual simulation of the faculty's 'Agility robot'. This simulation allowed for convenient, efficient robotic testing. The outcomes of the project led to largely increased knowledge of ROS2 and Gazebo. Additionally, a developed, but incomplete, simulation of the Agility robot had also been established.

In general, this method of robotic simulations and testing has large potential for the robotics industry. However, the main obstacle was the coding of custom plugins as they require in-depth coding knowledge and time to create.

Contents

Figure list

1. Introduction & Context

Modern progress in computational power and advanced computer software has led to increased access to accurate simulations. Consequently, simulation data now has the potential to produce reliable data without the application of real-world experiments and tests. This practise has been implemented into many sectors of engineering. For example, Computational Fluid Dynamics (CFD), Finite Element Analysis (FEA) and now the field of robotics can simulate robots and mechanical systems with respectable accuracy by use of 'in-the-loop' simulations.

In-the-loop simulations breaks down a real physical object, such as a car or robot, and re-builds it in a virtual environment. The optimal design aim of these simulations is to recreate the object's properties and functionality to a high accuracy so that the difference between the virtual and physical worlds are negligible. As a result, the testing data would be just as reliable as any tests in real life. At this point in time, in-the-loop simulations are not completely accurate but are good enough to mimic a real-world response to a high enough degree such that their results can positively influence design decisions and engineering judgement for its real counterpart.

In the context of this project, a 'Car-in-the-loop' simulation is required to simulate one of the faculty's robots. The robot is known as the 'Agility Robot' which is a robot retrofitted out of a high-performance, monster-truck, remote control car known as 'traxxas monster truck' and is in development to eventually become a fully autonomous, self-driving robot. An accurate simulation would aid development efficiency of the Agility robot by providing a useful tool that finds potential improvements to the robot's mechanical design, self-driving algorithms, and autonomous responses. Furthermore, simulating the robot's response is faster, cheaper, and safer than putting the robot in a real test. For example, if the robot crashes the simulation can simply be reset rather than fixing the damage physically. Additionally new parts can be tested to see if they have the desired outcome before they are bought for the real product. During one of the tests of the Agility robot it disconnected from the WIFI and nearly drove down a set of stairs before it was picked up, this would have caused significant damage. Incidents like this proved there is a real need for the in-the-loop simulator, especially because accidents and failures will always occur in a developing system. Also, any number of simulations can be produced at the same time. This means multiple users can generate data at once instead of waiting for data from a demonstration of the real robot which will take a significantly longer time. As a result, each member of the team can have their own version of the simulation to test, develop and optimise their own specialised areas of the robot which can then be brought together as sub-systems to generate a very advanced robot in the fraction of the time. Overall, the final simulator design will give future developers a tool to improve the quality and speed of their work by producing useful simulation data and reducing setbacks in progress which is important for any working environment.

Creating the simulation to a level of accuracy suitable for the agility robot's application would take a considerable amount of time. Consequently, the progress reported is intended to be built upon by future students and developers. The content of this report covers the beginning of the project and the first year of the simulation's development. The objectives for this academic year are the following:

- Provide a literature review for suitable Car-in-the-loop simulations.
- Design the beginning of a Car-in-the-loop simulation for the Agility robot.

There is a high demand for in-the-loop simulations in the engineering, automotive and robotic sectors. Due to increases in mechanisation in recent years the development and introduction of robots can be found in multiple job sectors on a global scale. For example, robotic arms on production lines, agricultural upkeep, autonomous driving and customer service. Although robotics is a relatively new phenomena, the new demand means in-the-loop simulation are needed to ensure robotic developments are efficient and safe. After the completion of this project, the knowledge gained could be used to help implement simulations for industry applications. In consideration of the Agility robot, the experience gained directly links to autonomous cars in the automotive industry and means if the project is successful the findings could be applied to increase autonomous safety and accident rates.

2. Research Background

ROS2 and Gazebo

The leading programs for robotic-in-the-loop simulations are ROS2 in tandem with the Gazebo Simulator. ROS2, also known as the robot operating system, is an open-source middleware that contains code libraries and tooling that aids robot creation, control, and movement. As ROS2 is open source, it means communities are creating code and uploading it such that anyone can use it. This makes robotic development much easier, especially for users with limited coding knowledge. Gazebo is a simulator which aims to generate accurate physical properties of a real-world environment. (OSRF-physics, 2014) The simulator makes use of four different physics engines known as Open dynamic engine, Bullet, Simbody, Dynamic animation and Robot toolkit to create the realistic setting. ROS2 is well integrated with Gazebo such that a robot created in ROS2 can be easily simulated with accurate real-life functions in Gazebo.
ROS2 and Gazebo are very powerful tools to create the working actions of simulated robots. As a result, it makes sense to use them to recreate the Agility robot in an in-the-loop simulation.

ROS2 API and tooling

The ROS2 tooling which are utilized to create useful representations of a robots mostly make use of nodes, topics, and services. Nodes are responsible for completing a single task. They are a container of code which completes a single functionality. This function has massive versatility but can be used to improve the robot, simulation environment or to launch required files for the simulation. Nodes that are responsible for fulfilling a robot's function usually come as groups of nodes in the form of publisher and subscribers. Publishers create information, sends the information on a topic which a subscriber can then connect to and create an action based on the information given. A topic is the way nodes communicate and can be considered a live stream of data which carries information in real time. The last important tool is services, which create information on topics and can directly influence subscribers with an intended goal sent via a topic. More complex functionality can be created when multiple nodes and topics are working in parallel with each other, plugins are a good example of this which will be explained later. From these tools a simulated robot can become highly complex, with accurate kinematics that are precisely controlled.

Ubuntu Operating System

As ROS2 is a middleware, its tooling is accessed from the operating system's terminal, in this case the Ubuntu terminal. Working from terminal requires its own set of skills as all its functionality uses terminal commands. Learning terminal commands is needed to work efficiently on many coding projects and especially for ROS2. For example, the terminal can be used to navigate through packages, 'listen' into topic and launch ROS2 nodes. Consequently, the terminal ubuntu commands should be learnt to aid the efficiency of ROS2 project

development. There are a set of commands to work in Ubuntu, while ROS2 has its own set of terminal commands that are downloaded in its installation. These are essential and can be found in Ubuntu tutorials and ROS2 documentation (ROS2Docs, 2021) (Linux, 2022).

Launch files

Working on a project for a significant time means that robots can quickly become complex systems that contain a multitude of nodes, plugins and movement controllers that help create a realistic and operational robot. Furthermore, the project should include a working environment and simulation. Inevitably launching all aspects of the robot may become a time-consuming process, especially when working from the computer's terminal.
This presents a need for launch files. Launch files are lines of code which can automatically launch all features of a project's robot, environment and simulation which reduces the start-up time of any project. Once a launch file is complete and provides all the information for the intended launch, it can be executed in the command line for ease of use. Launch files are created in the python programming language. ROS2 can read python launch files and has a command to execute them from the terminal (ROS2DocsLaunch, 2021).
Launch files are a good example of how ROS2 tooling is integrated with the system terminal to increase efficiency of code packages and projects.

Robot Description files

Robots are created using robot description files. These are responsible for defining all the characteristics of the robot and will describe the look, shape, physical properties and degrees of freedoms that can be turned into a visual, realistic entity when in the simulator.

During the project research it was found that there are two accepted types of robot description file known as Unified Robotics Description Format (URDF) files and Simulation description Format (SDF) files. The URDF file was created with the early versions of ROS and was used to define robots. However, as time went on, the URDF file lacked properties required for more demanding simulations. For example, friction and damping. As a result, the SDF file was developed for use in Gazebo and to make up for the shortcomings of URDF. As these two files had been created for different applications there are slight differences with each description type. At this current time, both SDF and URDF have experienced many updates and are now both highly compatible with features of ROS2 and Gazebo. In the context of the Agility robot the SDF file was selected as it is more current, created with Gazebo in mind and has more variety in its robot parameters. In some cases, URDF may be chosen because some ROS controllers only work for URDF files(OSRF-URDF, 2014).
At this point in time both file types are completely viable for in-the-loop simulations but have slight differences because of their origin.

There are two main components to robot description files which are links and joints. Links are the different parts of the robot while the joints are how the links connect and interact with each other. Every link has their own set of parameters that can be coded such that they respond to the physics simulator appropriately. Physical parameters include the link's mass, inertial axis and position of the part within the robot. Visual parameters include the colour and shape of the link. Finally, the collision parameters create the collision body of the link, this can be considered surface area of the link which also includes parameters such as friction and coefficient of restitution. Once all these parameters have been coded to match the real counter part's specification, links will act accurately in the gazebo simulator.
As previously stated, joints are used to describe how different links interact with each other by characterizing their degrees of freedom. Joints are a very useful tool for creating moveable robots. Once a joint is created they can act as nodes, in the sense that they can move or have forces applied to them when they subscribe to a topic with a data stream. This instructs the

joints how to react. Subsequently joints can have complex dynamics which can be controlled efficiently with the use of ROS2 controller nodes and plugins. Much like links, the joints also have a set of parameters that should be defined with the intention of creating joints that act like real-life degrees of freedom. The main parameter is the type of joint. Options include continuous, revolute, revolute2, gearbox, prismatic, ball, screw, universal or fixed joints (OSRF-Joints, 2020). Depending on which type of joint is selected, there will be a different set of optional parameters which effect the joints movement. Some examples of parameters involve damping, spring stiffness, friction, position of joint, which links are involved in the joint, movement limits and many more. Once all the joints of a robot are created, simulators visualise and produce joints and links with relevant physics based on their parameters in parallel with environment physics.

Integration of mesh files

Robot description files have limited capabilities when reading complex geometries. Many robots have been created using simple spheres, cylinders and cubes that can be scaled to a certain size. This is because robot description files must understand the code used to generate these shapes in the Gazebo simulator. However, this is not suitable for a complex robot which includes custom shapes.

A method for generating non-uniformed and complexly manufactured shapes in Gazebo is to use mesh files. Mesh files can be created using software such as blender and can be incorporated into a link's collision and visual parameters to give it physical shape in the simulator. An example of a 3D mesh file has the extension .dae (Lifewire-DAE, 2022). Robot description files can understand these files as they are written in XML which is an established mark-up coding language that SDF and URDF robot description's incorporate into their own files (Lifewire-XML, 2022).

The Gazebo physics engines use iterations in its simulations, so it's important to consider that having very small, intricate meshes are more computationally expensive to simulate than generic boxes and spheres.

Plugins and Controllers

After all the links and joints have been coded correctly with the intended parameters, a robot will exist as a convincing real-life imitation but will remain static in the simulator. Robots are usually created with practicality in mind instead of remaining still. Controllers and plugins are the solution to this problem. They implement movement and control in the robot's joints, allowing them to perform more enhanced dynamics.

Gazebo plugins are packages of code developed for use in the Gazebo simulator. There are six different types of Gazebo plugin which make a simulation more complex/functional: world, model, sensor, system, visual and GUI plugins (OSRF-Plugins101, 2012). World plugins are used to control properties in the world such as engine physics and lighting. Model plugins are used to control a robot's joints and movement. Sensor plugins create a sensor that gathers information, captures live video or sets sensor parameters. System plugins change the way gazebo simulation runs. Visual plugins can capture images from the simulation. Finally, GUI plugins can alter Gazebos overlay to display different tooling as desired.

Gazebo plugins are very similar to ROS2 nodes and due to their compatibility, plugins that exist in a simulation exist as their own nodes in the ROS2 system. As a result, ROS2 nodes and gazebo plugins are effectively indistinguishable from each other.

The main type of plugins/nodes used when dealing with robots are the model plugins, these are implemented into the simulation by attaching the plugin to the robot's URDF or SDF file. As these plugins are compiled as shared libraries, ROS2 and Gazebo have access and understand how the plugin works in order to make the robot move, sense and be controlled as intended. A package known as 'gazebo_ros_pkgs' has been created to ensure its shared libraries are compatible with both Gazebo and ROS2 and contains a variety of plugins that aid the creation

of a complex robot without the user having to code their own custom plugins (OSRF-stdpackage, 2014) (ROS wiki- gazebo_ros, 2015).

Sensors

Sensors are used in robotics to collect information on the robot and the environment. The information obtained by sensors are usually used to create feedback loops with the robot's functionality (Wikipedia – Sensors, 2022). Sensing greatly increases a robot's complexity and are vital in creating robots that react to their surroundings. The same methods can be employed into ROS2 and gazebo. There are several sensors that are officially supported by ROS2 but there are also many community-made sensors that may fulfil more specific problems. Sensors exist for position tracking, joint speed, joint force, audio recognition, 3D imaging and laser scanning which can be implemented by adding the sensor plugin to a robot description file (ROS Wiki-Sensors, 2022) (Contruct- Sensors, 2020). Instead of being controlled by topics, sensors produce information that are sent along their own topics. The other functions of the robot can subscribe to the sensor's topic to obtain a live feed of numerical data. Depending on the intended action that plugins and nodes are meant to fulfil, the sensor topics can be subscribed to by the plugins, interpreted and can react to the data stream provided. Subsequently, the plugin has a constant stream of information that allows for feedback responses and allows a robot to move in response to its surroundings.

Package Creation

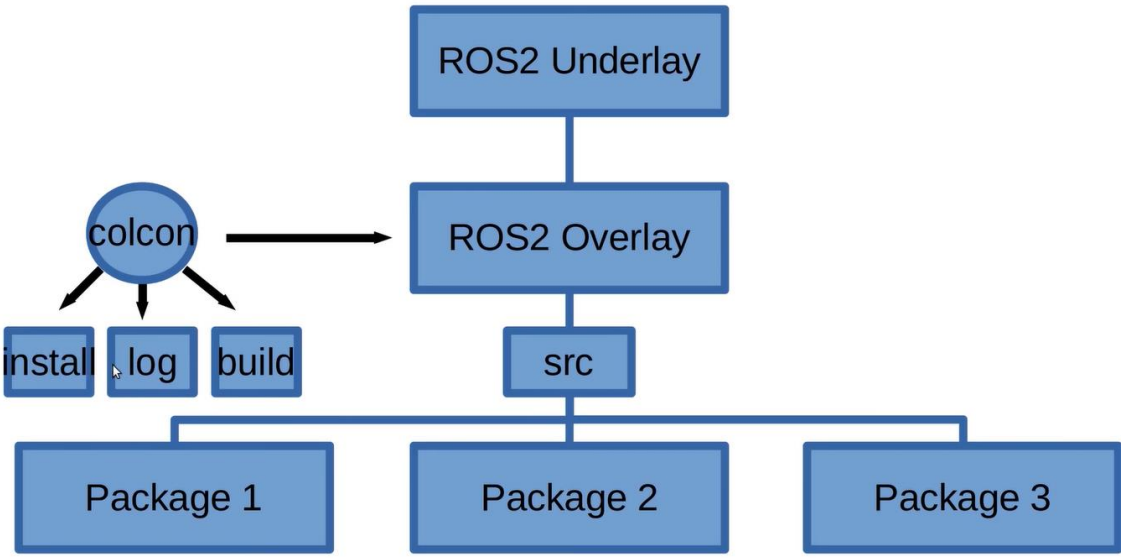
A package is a collection of files and code which have been organised such that a developer can work efficiently but also means the package can be transferred to other developers as a single container. ROS2 projects are stored in the form of a package or multiple packages.
To create a successful a ROS2 package that not only operates correctly but is also comprehendible for other users, all files need to be organised into the correct structure/architecture.

The overhead directory of the package is called a workspace, also known as a ROS2 overlay. This is a directory into which all other files and directories should be placed. The purpose of the workspace is to separate your own files from the files that are generated automatically with the ROS2 installation. Consequently, when coding any mistakes won't create issues for the base installation of ROS2 or other applications on the working machine. Furthermore, a workspace is a convenient area that contains all project progress. This is very important if there are many people working on a given project or if package needs to be sent, moved or uploaded at any point in its development (Robotogeddon, 2020).
Inside the workspace it is good practice to create a source folder named 'src' in which all the packages are located. A project is usually created in many different packages that reside inside the source directory.(Robotogeddon, 2020) (ROS2Docs, 2021).
Once a package is created inside the source file, the package will automatically generate some files which are necessary for the package to function correctly. These include the 'Cmakelist.txt', 'package.xml' and 'include' files. The Cmakelist is a file that contains code providing instruction for the building process of the package as well as all the libraries required and information about the package structure. The 'package.xml' file contains generic information which includes the package name, a package description, the developers involved and any other information the developers want to add for clarity for the user. The 'include' directory is where any header files are located if they are used in the package. Header files are used and required by many other files to work correctly therefore they can be accessed in the include directory.
In addition to these three entities, the files used to define the task and function of the project are located within the package. If desired these files can be located in another source file to keep the main bulk of the package separate from the essential files known as the Cmakelist.txt, package.xml and include directory (Robotogeddon, 2020) (Automaticaddison, 2021).

Package Building

When a package needs to be tested the operating system needs to understand how to run the package. As a result, a build tool is needed to call the required libraries and compile the entire package. This allows the operating system to run the package in tandem with the ROS2 tooling and with all the required code libraries.

Colcon build is the name of the build tool used for ROS2 packages. When Colcon is used to build a package, it will generate three more directories within the package's workspace. 'install', 'log' and 'build'. The install directory includes all executables in the package which allows the user to run executables in the command line with ROS2 tooling. The log directory includes all steps Colcon used to build the package. Finally, the build directory contains the intermediate files which are a product of building the package but don't have a significant purpose. (Robotogeddon, 2020)



(Robotogeddon, 2020)

Figure 2.1 – Diagram showing the standard format of ROS2 projects

History of ROS and Compatibility

The first version of ROS was created in 2007 and has since become a detailed tool for robot development which has used open-source communities and coding libraries to accelerate its growth. However, after developing said version of ROS, also known as ROS1, it was found it lacked many important features for wider use and industrial applications. The main problems involved real time responses and security. Consequently, ROS2 was developed from scratch, to make up for the flaws of ROS1 (Ros development, 2020).

At this point in time ROS2 is a relatively new application, and some may argue ROS1 is more useful as it has more documentation, plugins and more custom nodes created by the community. Nevertheless ROS1's end of life is scheduled for 2025 and means that ROS2 is the future for robotic development.

The development of ROS2 has meant much of ROS1's power and plugins are still being developed for use with ROS2. As a result, ROS2 has a few incompatibilities that will only be fixed over time. These incompatibilities are exacerbated by the integration of python3 and the Gazebo simulator with ROS2 which ROS1 developers hadn't considered at the time. Some documentation, research, nodes and plugins are not consistent for use in ROS2 which is inconvenient especially if desired plugins are created with ROS1.

The main differences between ROS1 and ROS2 are the integration of upgraded programming languages as ROS2 uses python3 and C++11 rather than python2 and C++03. Node writing has

changed as there is no transmissions or ROS master in ROS2. Launch files have been changed from xml language to python3 to increase their complexity and capabilities. Finally, building packages now uses Colcon rather than catkin which creates a large change between the layout of the Cmakelist in ROS packages and ROS2 packages. Furthermore, it means ROS1 packages can't be built using the ROS2 without re-coding their Cmakelist (ROS1 ROS2 changes, 2015) (Ros development, 2020).
If there is any case where ROS1 features are necessary for a robot working operation a package known as 'ROS1 bridge' can be used. This package provides a system which enables messages to be sent between ROS1 and ROS2. However, as the end-of-life date for ROS1 has been announced its evident that it's not good practice to rely on ROS1, especially if future projects like the Agility robot simulation may still be used or in development after the end of ROS1's life. All things considered ROS2 is much more powerful than ROS1 and should be used for any new robotic project even if some plugins are unavailable.

GitHub
GitHub is a website which provides a platform for communities to upload their code packages and projects. Its main advantage is easy sharing of coding projects. It's also used to allow teams to develop a project at once in a single place and allows for live updates. As a result, it's a very useful tool for developers to share projects and work on projects together. There is strong ROS2 and Gazebo community on GitHub that produce custom nodes and plugins that vary in function. Depending on what users want to create or what is beneficial for the community there are people who use their own coding skills to create packages that help others.

Auto Rally
Auto Rally is a project created by Georgia Institute of Technology (Georgia Tech) and aims to develop highly aggressive autonomous algorithms which allow robots to drive at speeds of 20mph while only being controlled by sensor information and self-driving capabilities. The institute uses a modified remote-control car with extremely advanced functionality. The Georgia Tech team have designed their robot with the help of ROS developers to assist in the project's engineering decisions but also to see if the ROS community can improve their autonomous driving controllers.
Auto Rally is similar to the Agility robot project, in the sense that it's a car-like robot intended for autonomous driving. Consequently, their research papers were read to gain an initial understanding of ROS and how it can be used in a car-like simulation. The Georgia tech project has been created on ROS1 and intended to be used with the robots hardware. Furthermore, it is in a very advanced stage of autonomous driving and means many complex custom plugins and coding have been created which are not applicable to the current stage of this project. However, reading the project and understanding its origins provided strong inspiration and a good baseline of robotic knowledge. Additionally, the GitHub repository created by Georgia Tech helped understand the working principles of GitHub (AutoRally, 2020).

3. Project Progress

Once extensive research had been achieved the knowledge gained could be applied to start the agility robot's simulation. The first action needed was a suitable workspace and a correctly configured package.

A workspace named 'robot_description_ws' was created with a source folder named 'src' inside the workspace. Inside the source directory a package was created using ament_cmake commands, this packaged was called 'agilityrobot_description'. Using ament_cmake to generate the package meant the package was created with an include directory, Cmakelist.txt file and a package.xml file which was automatically generated. Further directories were created in the package to keep the package organised with all code that would eventually complete the project. Using examples of previous ROS2 packages on Github for guidance, directories by the name of 'launch', 'sdf,'urdf', 'xacro', 'rviz', 'config' and 'README' were added to the package. This base structure meant all created files would have a suitable storage location and if any directories ended up unused by the end of the project they could just be deleted with ease. After that the base code structure of the Cmakelist.txt was taken from Github and the file was edited such that it was suitable for the agility robot package so that Colcon build could build the package successfully. This included specifying the minimum required versions of the building tool and coding languages. Furthermore, the dependencies for libraries required to build the package were included in the Cmakelist.txt. Finally, all directories needed to be installed so they could be accessed from the command line. This action is defined in the Cmakelist.txt by specifying which directories are in the package so Colcon knows what to install and build. As a result, the directories were defined inside the Cmakelist.txt, if any other directories were created during the package development they needed to be added here. Using the command 'colcon build' from the terminal the package could be built successfully after fixing a few syntax errors.

Now that the package could be successfully compiled, arguably the most important and time-consuming file needed to be created. The robot description file.
The first step in developing the agility robot's robot description file included making detailed CAD models of each part of the agility robot. The purpose of making models for each individual part meant the surface of the CAD models could be converted into a mesh that could be added to each link's collision body and visual parameters such that each part of the robot looked but also had a collision body identical to the agility robot. Consequently, the robot would interact with objects in its environment and collide with them as the shape of the agility robot. Models of the robot's chassis and wheels were created from scratch with dimensions taken from the official Traxxas website and the physical agility robot. Thus, all critical parameters such as the front wheel track, rear wheel track, wheelbase, wheel radius and wheel thickness were re-created accurately.

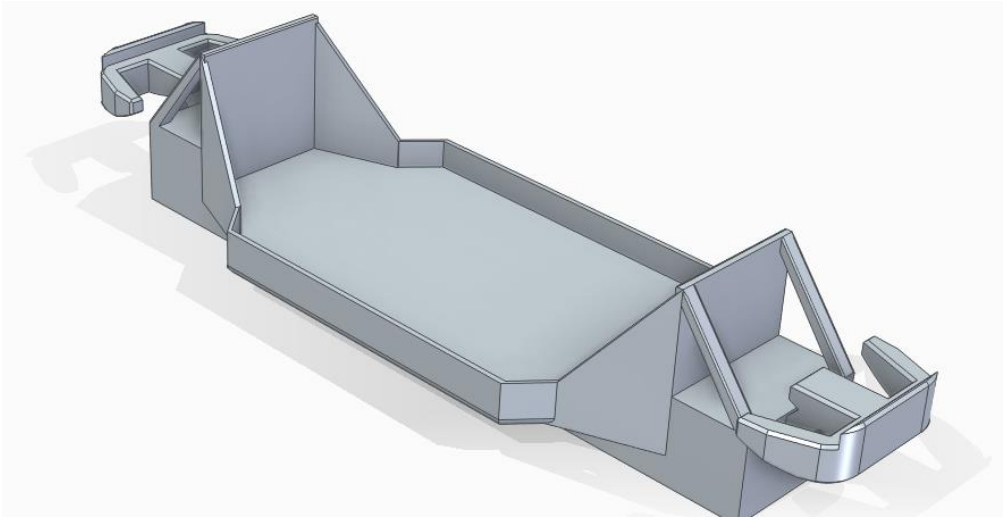


Figure 3.1 – CAD model of robot's Chassis.

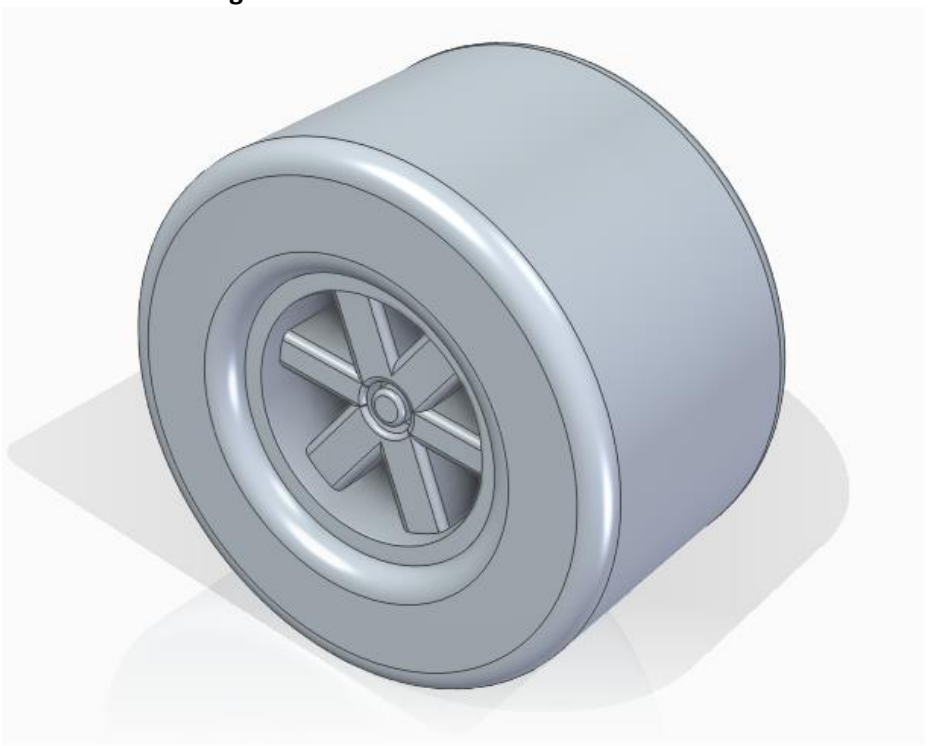


Figure 3.2 – CAD model of robot's Wheel.

Other parts of the robot's models had been previously created by another student, these included parts of the lidar sensor, front camera and electronics control housing.

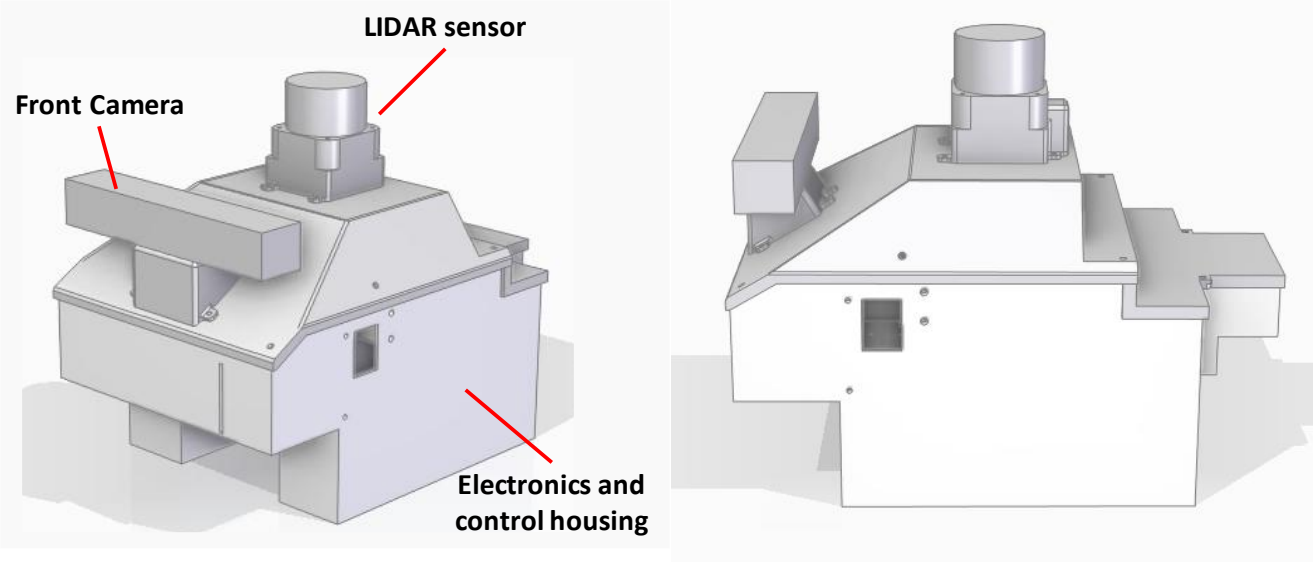


Figure 3.3 – CAD model of assembled Lidar sensor, camera and electronic housing.

The chassis, lidar sensor, front camera and control box were then assembled in a solid works assembly such that they were all placed correctly in relation to each other and could be found in a single file. After all the models and assemblies had been completed all files needed to be converted to a file type that the robot description file could read and generate a mesh from. Therefore, they were converted into STL files. STL files were a version of CAD file which presented the geometry of each part but didn't include any physical information. For example, the part's material, mass and inertial axis. However, this was a needed step because the files could then be placed into blender and converted into .dae files. SDF robot description files are compatible with .dae files to generate a mesh that can be used as a collision body and visual mesh. This process meant the files had no physical parameters and meant they needed to be added separately in the code of the SDF file. Nevertheless, the meshes still allowed for a good quality visual representation and accurate collision geometries to be added to the robot's SDF description file that could be simulated in Gazebo. After that, this process was repeated for the parts of the robot's double wishbone suspension.

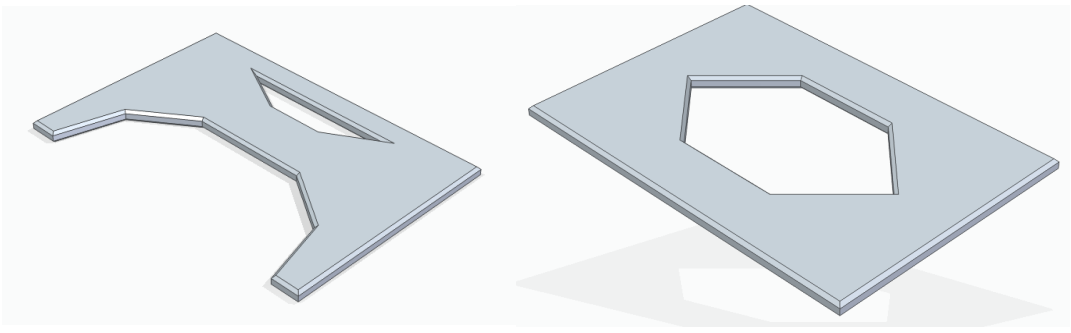


Figure 3.4 – CAD models of double wishbone suspension top and bottom struts.

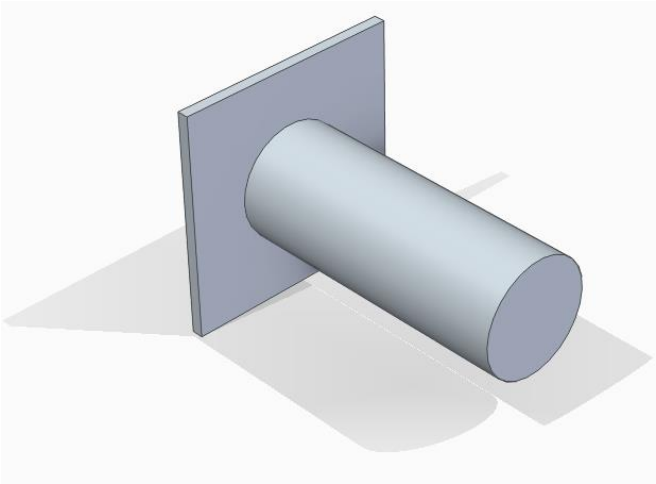


Figure 3.5 – CAD model of suspension-wheel spindle.

Now that the meshes had been created, the following action required all parts to be defined inside the robot description file. The meshes could then be added to their corresponding parts and each part could be completed with their coded parameters.

As previously stated, each part is defined as links in the robot's description file. The first link was created and named as "chassis". The physical parameters were then coded by pasting the code structure for all needed physical properties into the chassis' link tags. At this point the magnitude values for each physical parameter could be added in the mandatory lines. First the mass value was added. During the development of the agility robot the mass had not been accurately tracked and the total mass of the chassis and control box with all its electronic components inside was not specified. However, after attending some agility robot demonstrations the robot had been picked up and an estimation for this part's mass was coded as 25kg. The mass' inertial axis were added and the position of the link was selected to be at the origin this was a justified location as many other links would be positioned in relation to the position of the chassis. Other physical parameters included gravity multiplier, enable wind, self-collide and kinematic. These were left as default such that the gravity experienced on this part was the same as Earth's gravitational pull while the other parameters were unnecessary for the purpose of this project. However, its useful to know these options can be changed if a different dynamic or working environment wanted to be tested on the agility robot. This concluded the code of physical parameters for the 'chassis'.
Next a set of visual parameters were needed for the 'chassis'. A visual tag named 'visual_chassis' was made which is where all the visual characteristics for the chassis link would be located. Here a visual position was defined at the origin of the chassis' origin which makes sense as the link should materialise in the simulator in the same position as it is physically acting. After that, the previously created mesh of the chassis and control box was classified as the link's geometry and the colour of the mesh was selected by coding in the relative path to the desired colour file inside Gazebo's colour libraries. Dark grey was selected as it was similar colour to the aluminium control housing.
The final part of link creation requires the collision parameters to be defined. A collision tag was created and named 'collision_chassis'. Inside this tag the collision body was defined by the chassis' mesh this meant the faces of the link take up the shape of the mesh and act as a physical body in the simulator. Other parameters included in 'collision_chassis' are surface friction and coefficient of restitution. Low values of friction and no restitution were coded such that the link would act as a solid, low friction body. As the chassis is mostly made of stiff plastic and aluminium these parameters were justified. At the end of the tag some parameters are required by physics engines 'ODE' and 'Bullet'. They were coded such that the collision body acted as a solid body.

This process was then repeated for each individual part of the robot. The code structure used for the chassis' link was reused for each of the other links, but the values were updated in correspondence with the specific real-world part. Furthermore, the appropriate mesh files were coded into each link's visual and collision parameters.

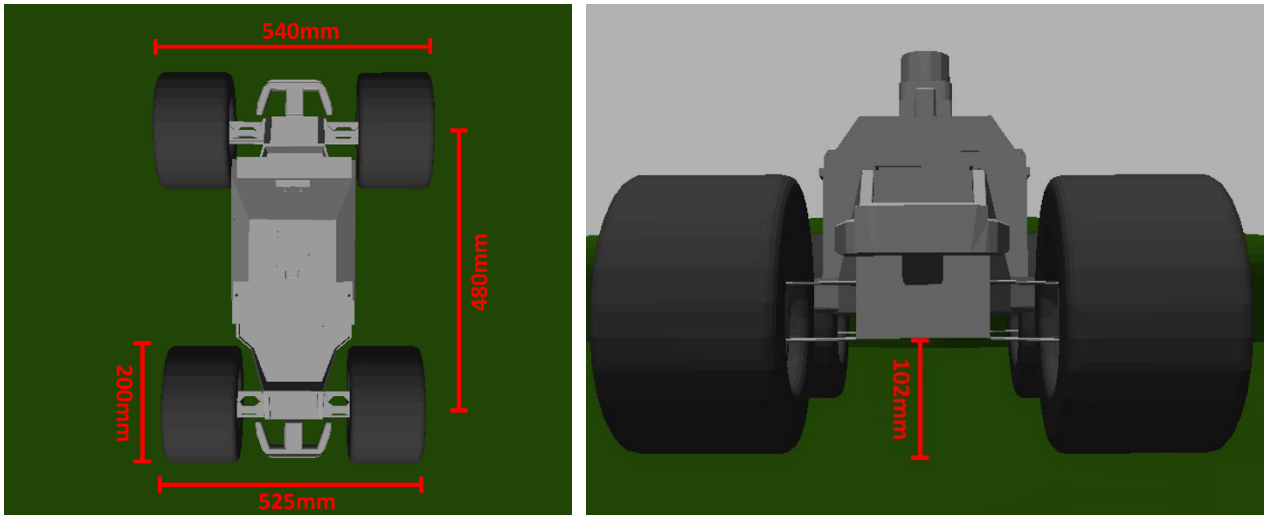


Figure 3.6 – SDF model in Gazebo simulator with wheel track, wheelbase, wheel diameter and ground clearance presented.

The final SDF version included 17 links, thus an absolute path for a corresponding mesh file can be found 34 times to define each link's collision geometry mesh and visual geometry mesh. The meshes had been created to achieve the desired vehicle parameters specified on the traxxas website. These are shown in figure 3.6. Using the position parameter in the link's physical section the links could be placed into this arrangement which resembled the real agility robot. However, at this stage of the project the robot would fall apart when Gazebo simulator was started. This is because the links hadn't been attached to each other but rather suspended in space next to each other. As a result, the joints needed to be specified where the links should be connected. This would form a more complex robot with the desired degrees of freedom and interactions between the different links.

To explain the code structure of a joint, an example joint between the chassis and the font, left, bottom control arm will be recounted with its parameter choices explained. This is because all joints follow the same structure but may have different parameters based on its purpose.
First the joint name and joint type had to be defined. This particular joint was named 'chassis_FLBwing_joint' this followed the same naming structure for all the joints and was appropriate as it represented which links were included while also specifying it's a joint. The joint type was 'revolute', Gazebo recognises a revolute joint and knows the two links will rotate around each other at a defined point. After that, the chassis was defined as the parent link while the front, left, bottom suspension wing/control arm (FLBwing) was defined as the child link. This meant the FLBwing would rotate around the chassis instead of the chassis rotating around the suspension wing. Next the position of the joint had to be specified in relation to the child link. The axis of rotation was also specified, revolute joint cans rotate around the x, y and z axis but can also rotate around a custom axis using a superposition of unit vectors that point in the x, y, z directions. The 'chassis_FLBwing_joint' was coded such that it rotated around the y axis. Limits of rotation, velocity and torque can also be set on a joint which wasn't necessary for this specific joint and was left as the default values. Finally, parameters defining the rest spring reference position, spring stiffness, damping coefficient and friction can also be specified. These were also left as the default zero values.

This process was then repeated for 22 different joints which completed all the desired degrees of freedom between all joints within the SDF file.

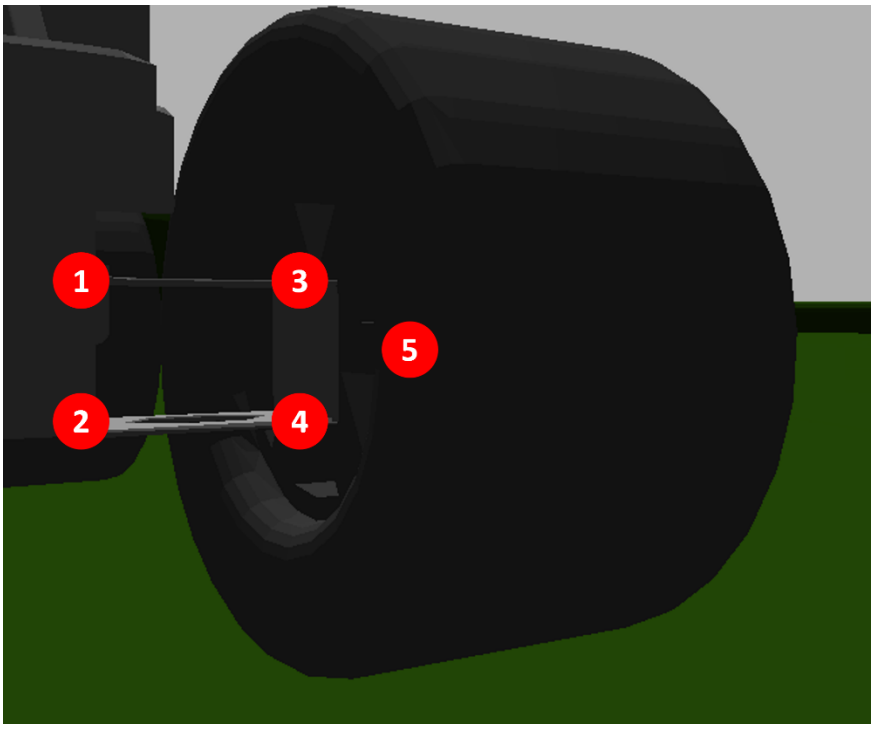


Figure 3.7 – Figure showing location of joints at the front left section of the robot.

The joints presented in figure 3.7 are represented by the numbered point. These show the rough locations of the joints. Points 1, 2, 3 ,4 ,5 are named 'chassis_FLTwing_joint', 'chassis_FLBwing_joint', 'spindleFLT_joint', 'spindleFLB_joint' and 'wheelFL_joint' respectively. Joints 1 and 2 allow the control arms to rotate around the chassis and joints 3 and 4 allow the wheel spindle to rotate around the control arms such that it keeps upright and only moves vertically. Finally joint 5 moves the wheel around the spindle which moves the robot.
All the joints are revolute joints and makes up the front left double wishbone suspension which aims to accurately imitate the dynamics of the agility robot's suspension. However, after testing the robot's static response in the simulator, it was found that the suspension wasn't acting correctly. The simulator showed the base of the chassis scraping the floor and the suspension couldn't hold the weight of the chassis and control box.
This problem occurred because there was no spring stiffness offering suspension to the design. There were two possible solutions to this problem. Solution one; add another link which would act as a spring between the bottom suspension wing and the chassis. The second option included adding a spring stiffness to the joint that connected the bottom suspension wing to the chassis. The first option is the most accurate to the real robot which suggests it's the better option however the second option was opted for. This is because the same goal can be achieved with less work. The reason the robot's real counterpart has an entire spring-damper system inside its suspension is because its constrained by real world physics and requires these extra components to generate the desired suspension response. But in the simulation, we can add parameters that define damping coefficient and spring stiffness to a single small point (the joint position) which generates the same response without the need for these extra components. This shows that option two was the most beneficial as it saved time but still produced accurate results that are in line with the real agility robot.
One significant point with this option is that the spring stiffness acts closer to the chassis whereas on the real robot the physical spring was connected nearer to the centre of the bottom control arm.

As a result, the spring stiffness parameter on the joint may be a larger value than that of the real spring to achieve the same equivalent stiffness and therefore the same moment balance equation on the bottom control arm.
The spring stiffness of the Traxxas monster truck hadn't been specified on its website as a result there wasn't a reference spring stiffness to calculate an appropriate joint parameter. Consequently, the correct value was found through trial and error until the resting ride height was near the position on the Traxxas specification description and the suspension system acted similarly to the real robot as seen in the demonstrations. The final spring stiffness of the 'chassis_FLBwinig_joint' was set as 175N/m. The same value was adopted for the other double wishbones.
The suspension system was then tested in Gazebo to see if it had an accurate dynamic. This was done by manually moving the robot and dropping the robot from small heights. The results found that the spring stiffness was a suitable value that causes the suspension to act as intended. However, after undergoing a force the robot would experience prolonged oscillations causing the chassis to bounce. This was evidence that the suspension design wasn't dissipating energy as it should have been. This meant a damping coefficient was needed. A damping coefficient of 0.2 was added to the same joint which contained the spring stiffness and made the joint act as a spring-damper system on the bottom control arm. This is the same design as many suspension systems and is the same in the case of the agility robot. Further tests found that the given parameters gave a very convincing, desirable dynamic that was appropriate for mimicking its physical equivalent. Hence, the same damping coefficient was implemented into all other chassis-bottom control arm joints.

Now that the suspension design had been optimised it meant the steering design had to be created, such that an Ackermann style of steering could be replicated. The agility robot contained front wheel steering concurrent with four-wheel drive. As a result, the steering joints had to be implemented without constraining any driving forces. To do this it was decided that the wheel would still rotate around the spindle's single axis but the spindles themselves could be rotated to allow a steering motion. This meant steering joints were added to the front suspensions at the centre of the spindle's flat plate. These steering joints were named 'FL_steer' and 'FR_steer' and were defined as revolute joints that would rotate around the z axis. The 'spindle' link would connect to both double wishbone control arms so it can move vertically with the control arms but would be able to rotate around another axis to allow for steering. After testing this configuration by manually inputting a desired steering angle to each joint it was found that they weren't moving at all. It was later found that when a 'revolute' joint axis of rotation is defined it created restrictions on all other degrees of freedom. In the case of the suspension design the 'spindleFLT_joint' and 'spindleFLB_joint' had been defined to have an axis of rotation in the y axis. This restricted the spindle and didn't allow for rotation of the link in the z axis even though the steer joint had been defined to move that way. To overcome this problem 'spindleFLT_joint' and 'spindleFLB_joint' joint types were changed from 'revolute' to 'ball' which is the code syntax name for a ball and socket joint. This allowed the spindle to move in a 360-degree motion around the control arms. However, as the spindle was connected at top and bottom and had the steering joint controlling its motion in the z axis it means the spindle could only rotate in the desired planes. These were around the z axis and rotate with the suspension control arms when they moved up and down.
The final result of these changes meant the suspension design was working as before but allowed for the spindle to rotate around the z axis, allowing for an Ackermann front wheel steering dynamic. The general position of the steering joints has been represented in figure 3.8.

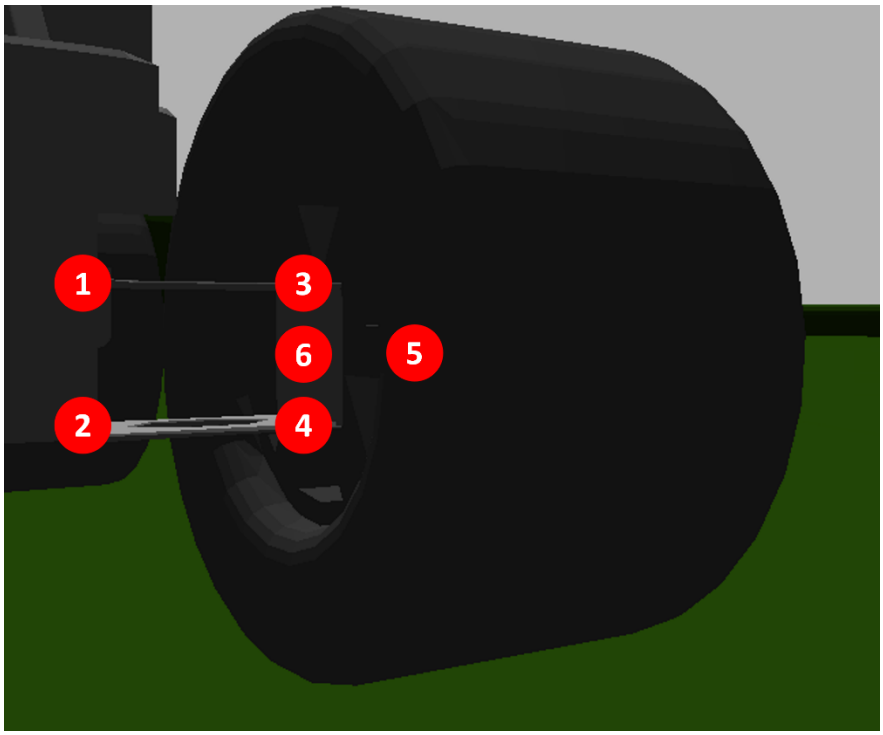


Figure 3.8 – Figure showing location of joints at the front left section of the robot after updated steering.

The steering joints worked well but tended to drift to random angles when the car moved forwards and backwards. Therefore, a small spring stiffness was added to the 'FL_steer' and FR_steer' joints such that the wheels point naturally point forward unless commanded to do otherwise.

At this stage the wheels acted like solid bodies which wasn't realistic. The working behaviour of the tyres are very important in making a vehicle move as they generate the lateral and longitudinal forces required to accelerate, brake and corner. However, tyres have very complex, non-linear responses which make them hard to simulate accurately. An accurate model of a tyre would incorporate parameters such as the friction ellipse and cornering stiffness however the SDF file does not have these parameters available.
Instead, each tyre links was given a coefficient of restitution which made them slightly soft and bouncy. Furthermore, their coefficient of friction was increased to a high level similar to that of treaded rubber. Additional parameters were also given values which helped the tyre's behaviour, these were contact patch radius and torsional coefficient. These parameters can be found within 'friction' tags that lie within the wheel links.
All these factors helped create more realistic tyres that the simulator could use to generate realistic tyre physics.

After completing all the links and joints with all their parameters the robot description file was very long. In fact, at this point the SDF file contained 2344 lines of code. If any parameters needed to be changed or inspected it would be hard to find, especially if the same names and numbers come up multiple times. This is inadequate presentation, especially if new developers want to continue the project and understand the SDF file. As a result, all sections of code were broken up under comments which made the file much more readable. The links were under their own section and all their parameters were contained within their specific link. The joints were in another section. Every link had their parameters contained within a section named the same as the joint itself. This made

each section of code for each joint or link easily accessible and generally helped the visual clarity of the file.

Now that the SDF file had been created such that the stationary robot had been fully defined, it was time to create an environment that the robot would operate and be tested in. To do this, CAD models had to be created which are then turned into a mesh and added to the simulation world. The purpose of having an environment is to aid robot testing by forcing the robot to manoeuvre and complete certain motions which would be completed in real life. If the robot can complete the motions in the simulation the real robot should be able to complete it in a real-world scenario too. Keeping this in mind, two environments were considered. The first environment included a figure of eight track on a grass surface. This was a suitable design as it would experiment with the robot's steering system, driving motion and tyre traction. The second environment design was based in an urban setting, it included a traffic crossroads with objects on the road that represented cars. This environment would be very effective in the robot's dynamics with its self-driving algorithms. In the end both environments had been developed in CAD software but the figure of eight track was selected to be used during this year of the project. This is because the beginning of the project wouldn't include any testing of self-driving algorithms. In fact, the real agility robot wasn't near to becoming self-driving yet so its would've been too early to start considering this aspect of the project. Nevertheless, this environment had been developed if it's needed in the future and the figure of eight track will be used for now.

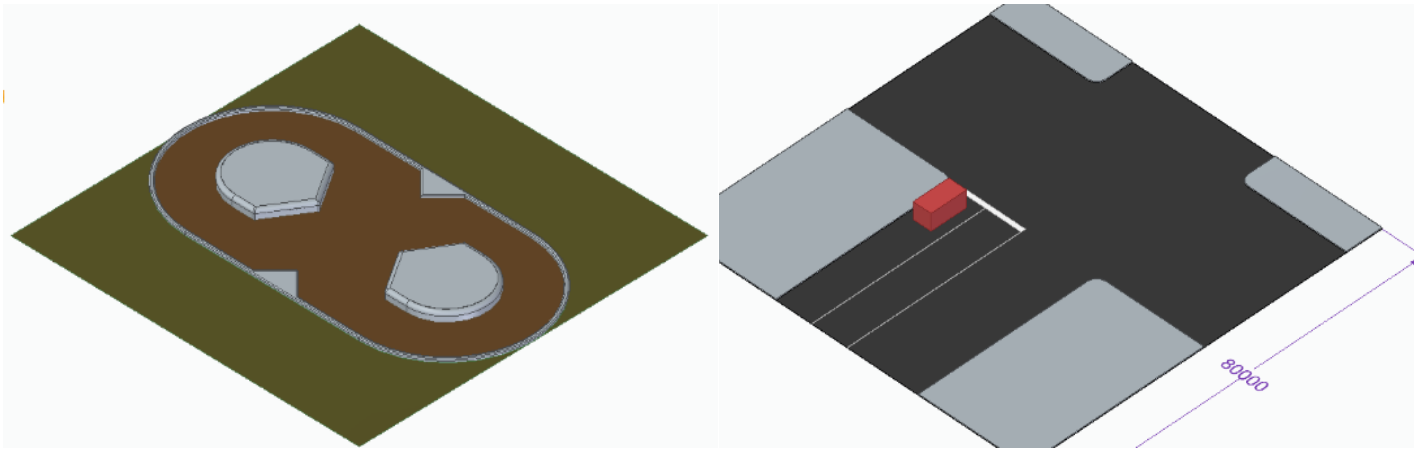


Figure 3.9 – Picture showing environment CAD models

A new SDF file was created to create the track. A single link was created and named '8track'. Inside this link contains the physical, visual and collision properties of a grass figure of eight track. The tracks mesh files made up the visual and collision bodies. The colour 'Grass' was selected from Gazebo's texture library.

From the agility robot's SDF file and the environment's SDF file a world was created. A world is a file which specifies what models should be spawned into Gazebo so that the simulation is fully complete. This world file included the figure of eight track and the agility robot by coding their absolute path in parallel with the position these SDF files should be placed. Addition parameters can be added to world files. In the context to this project's world file the track was stated to be static and there was also a global light source.

Launch files

A python launch file was created and named 'agilityrobot.launch.py'. The aim of this launch file was to create a single executable which would launch every aspect of the simulation needed to start a testing situation.

First the launch file needed to import some python libraries to work properly and such that some launch commanded were understood. These included libraries from ‘launch’, ‘launch.actions’, ‘launch.conditions’, ‘launch.launch_description_sources’, ‘launch.substitutions’ and ‘launch_ros.actions’. Although not all these code libraries are needed to run the launch file they are included for completeness and aid the coding of any functionality a launch file can achieve. Within this file it launches the Gazebo software and spawns the previously created world inside the Gazebo simulator.
This launch file was very beneficial for convenience and timesaving as the python file created an executable which ROS2 could read and launch with its ‘ros2 launch’ command. This made the start-up time of robot testing much faster. Furthermore, if any more nodes, plugins, robots or objects need to be spawned in the future of this project they can be placed into this file for convenience. This avoids launching them separately and ensure all aspects of the simulation that are needed are being launched when gazebo is started.

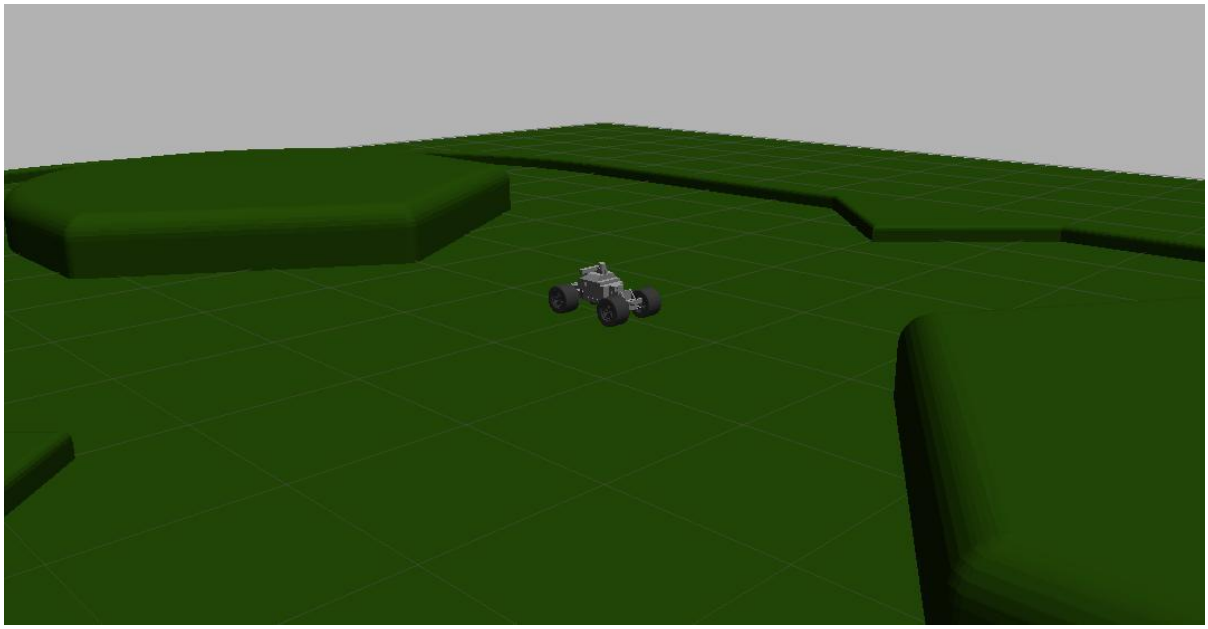


Figure 3.10 – Picture showing the world spawned with ‘agilityrobot.launch.py’

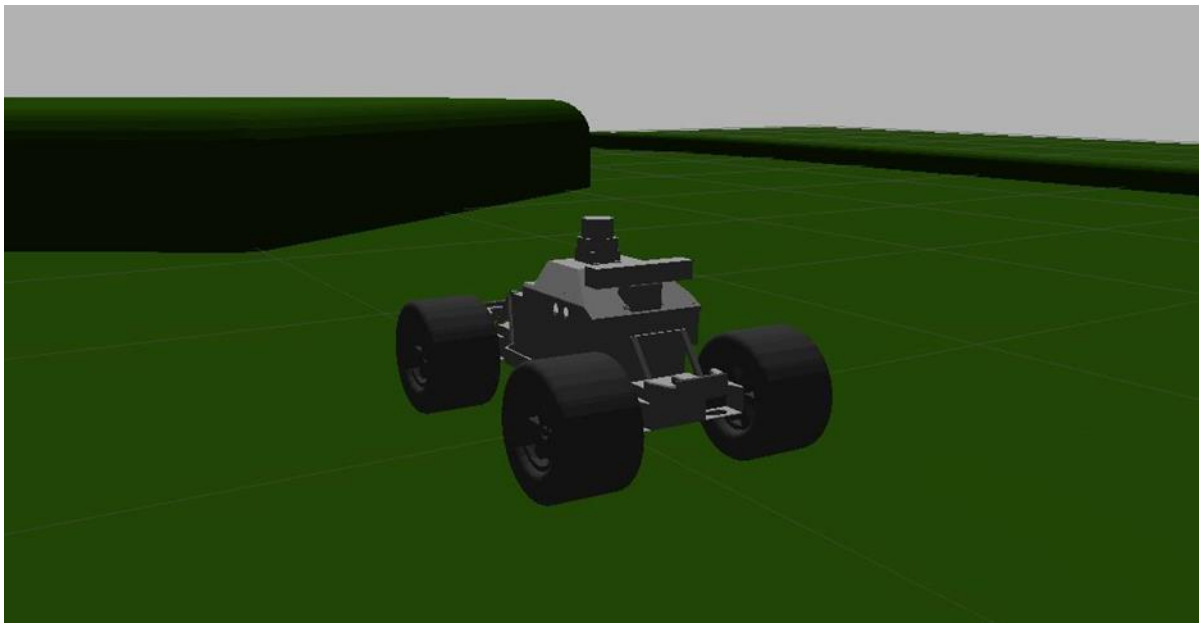


Figure 3.11 – Picture showing the robot resting in the world spawned with 'agilityrobot.launch.py'

To move the agility robot, plugins were needed to control its joints. The method used and plugins selected to move the robot's joints will determine how accurate the dynamics in the simulation test will be in correlation to its real equivalent. A package known as 'gazebo_ros' was found during the project research. This package contained a variety of plugins which had been used for ROS1 projects but had been migrated for use in ROS2 in tandem for use of Gazebo. As a result, this package was downloaded and used.
In order to use 'gazebo_ros' with the project, the package needed to be sourced with the simulation. This would give the simulation access to all plugins contained with this package which could then be used with the agility robot when specified. Thus, the package was added to the launch file by sourcing 'gazebo_ros' in the launch description.
Once the simulation had access to the package, the model plugins could be used by adding them to the agility robot's SDF with the required parameters/arguments.

The first plugin added to the robot was the joint state publisher. This plugin is used to keep track of every joint position, velocity and torque values which are produced as information on topics. As a topic is created for each joint it means the values can start to be controlled or responded to. For example, if the simulation intended the robot to move forwards, an intended velocity value could be published to the wheel's topics to create the intended motion. In contrast the values produced by the joint state publisher could be subscribed to and reacted to if desired. For example, if the speed limit is 5m/s and a controller obtained larger values from the robot joint publisher's topic it could lower the total speed. The functionality and examples are extensive, but the point is: information of joint states is vital for communication and data sharing across every aspect of the agility robot is needed to allow for feedback and control. This shows that the joint state publisher is a justified addition to the agility robot and necessary for its functionality.
The joint state publisher was added to the robot's SDF file by adding a plugin tag and specifying the plugin's name and file name. After that the parameters were required, this included the information stream's update rate. This was selected to be 5Hz. This value is relatively low to reduce computational effort. However, this value can easily be changed to a higher value if the information rate leads to unsatisfactory lag time in the robot's nodes/controllers in the future development. The update rate required depends on the application of nodes using the information published. From previous research typical values of the update rate for the joint state publisher are in the range of 20-60Hz. The second parameter needed for the plugin included which joints the developer wants to

publish values for. In the agility robot the four driven wheel joints and the two front steering joints were added. This is because these joints will be actively controlled in the future of the robots development. Their position, velocity and torque values need to be known and controlled to create practical, complex and controlled robot dynamics. The other joints in the SDF file, such as the suspension joints, were not added to the joint state publisher. This is because are passive joints that don't need to be actively controlled but serve their intended purpose solely on their previously defined parameters.

Now that the joint states were being published more plugins could be used to control the robot's movement using the topics existing from the joint state publisher. The aim was to create four-wheel drive and an Ackermann steering system. To get a driving motion that's true to the real robot, an intended driving velocity would be entered that the wheel joints would try achieving by use of input torques and PID controllers. Eventually its steady-state value would be the same as the desired input velocity with minimal steady state error. This approach was good as it meant the PID controller parameters could be tuned until the acceleration and torque characteristics were similar to the real agility robot.
Multiple plugins were researched to see if it fit the criteria. First a plugin from the 'gazebo_ros' package was called 'gazebo_ros_force_system'. This plugin published a service onto a joints topic which created a specified torque on a joint. However, this plugin wasn't a model plugin but a world plugin. Consequently, the service was created via terminal commands and only one service would be created at once. This was not acceptable for the agility robot as four topics needed to be subscribed to at once for the four-wheel drive system to work correctly. Furthermore, the plugin didn't have PID control. As a result, this solution had to be abandoned.
The next plugin was named 'Ackermannsteer'. This plugin was a custom plugin created by a user on Github. The plugin is for a four wheeled vehicle and produced rear wheel drive with Ackermann steering on the front wheels. Again, this was not the desired dynamics intended for the agility robot and therefore it wasn't used.
The final plugin had much more potential. It is part of the 'gazebo_ros' package and is called 'gazebo_ros_ackermann_drive' which has been developed for use with ROS2. This plugin creates a four-wheel drive vehicle with the front two wheels having the capability to steer. In addition to the wheel functionality, it has PID controllers to achieve the intended velocity in a realistic manner. As a result, this plugin was implemented into the robot's SDF file. To apply the plugin, the plugin's name and file names were defined inside a plugin tag. Following that, the parameters needed to be defined. The Ackermann drive plugin had several required parameters. The initial parameters defined the ROS2 topics which the input information would be sent on.

```
<plugin name='ackermann_drive' filename='libgazebo_ros_ackermann_drive.so'>

  <ros>
    <namespace>demo</namespace>
    <argument>cmd_vel:=cmd_demo</argument>
    <argument>odom:=odom_demo</argument>
    <argument>distance:=distance_demo</argument>
  </ros>
```

Figure 3.12 – Figure showing code that define the ROS2 arguments and topic names.

The parameters were wrapped in a 'ros' tag such that ROS2 understood the parameters were intended for use with ROS2 tooling. The first argument/parameter is the namespace. This groups all the topics under a collective heading such that the user knows that they are all part of the same plugin. This namespace was changed from 'demo' to 'ackermann'. The following parameters found in figure 3.12 define the names of topics that control the robot's velocity, robot odometry and total distance travelled which had default names 'cmd_vel', 'odom' and 'distance' respectively. The topic

names would be remapped using an equal sign within the argument tags. These topic names were remapped/renamed to ‘vel_ms’, ‘odom’ and ‘distance_m’. Consequently, these names will appear and can be used in the systems terminal once the simulation has started.
Next the update rate and joint names needed to be defined. The update rate was left as 100hz for a quick dynamic response. The name of the four driven joints and steering joints was defined into their respective parameters required by the plugin. As shown in figure 3.13.

```
<!-- wheels -->
<front_left_joint>wheelFL_joint</front_left_joint>
<front_right_joint>wheelFR_joint</front_right_joint>
<rear_left_joint>wheelBL_joint</rear_left_joint>
<rear_right_joint>wheelBR_joint</rear_right_joint>
 <left_steering_joint>FL_steer</left_steering_joint>
 <right_steering_joint>FR_steer</right_steering_joint>
<steering_wheel_joint> </steering_wheel_joint>
```

Figure 3.13 - Figure showing code that define the joints used in Ackermann drive plugin.

There was also a parameter to limit the linear velocity and steering angle. The max velocity was set to 10m/s and the steering angle was set to 0.26 radians as it was a good approximation to the real values. The PID parameters were left as the default values and the publishing values were set to true to gain some feedback information in the terminal.

This completed the structure of the Ackermann drive plugin, but after launching the simulation the plugin had no effect on the simulation. None of the topics and functionality could be seen or used. One possible solution to this problem could be that not all parameters are completed correctly. For example, the last line in figure 3.13 presented a potential problem. The virtual agility robot didn’t comprise of a steering wheel joint because its real counterpart didn’t have one. Nevertheless, the plugin required one to work. To fix this problem the source code of the plugin could be altered in order to take out the steering wheel functionality and customise the plugin to work especially for the agility robot. Considering the Ackermann drive plugin already contained most of the desired functions it would makes sense to retrofit this plugin instead of coding one from scratch. The first academic year didn’t have enough time to code a specialised plugin for the project. The research and ideas generated this year have been passed onto the next developer of the project. Consequently, a retrofitted Ackermann drive plugin could eventually be specifically developed for the project to get optimal robot dynamics.
Having four-wheel drive, Ackermann steering, PID control and torque joint input on each joint in a single plugin is a very specific, complex requirement. Consequently, no plugins fitting this specification was found during the research. This means a custom plugin will have to be developed at some point in the projects lifetime, unless a more appropriate one gets developed by the ROS2 community in that time.

At this point in time a differential drive plugin had been added to the agility robot. This is a very simple plugin that drives the front two wheels and steers by rotating the front wheels in opposite directions which caused the whole robot to rotate around an axis. The required plugins were added and worked successfully with the agility robot. Once the plugin had been added it could be used in tandem with a node by the name of ‘teleop_twist_keyboard’. This node uses the keys on the systems keyboard to publish information on topics. The teleop_twist_keyboard’s topics were remapped as the differential drive ones which allowed keyboard inputs to be sent along the plugins topics and used to control the differential drive in the simulation. This worked very well and was the first example of controlling the robot ‘remotely’ via topics. This also proved that the problems

surrounding the Ackermann drive plugin were with the plugin itself rather than in code syntax or the robot's SDF file.
Originally the differential drive plugin had been added to practice implementing plugins and to see how they worked. It's important to note that this plugin won't be used in the final version of the project because it doesn't accurately mimic the way the real agility robot moves. The plugin can be considered a temporary solution to robot's control before the Ackermann steering plugin is fully customised and integrated into the SDF file.

The last stage of this year's development required a 'README.md' file. This file is an explanation of the project to help setup a simulation without having any prior knowledge of the projects coding. This is very important, because the aim of the simulation is to allow the agility robot to develop efficiently. If the user required in-depth knowledge of the coded files to complete a simulation, then extensive reading and coding experience would be needed which would waste significant time.
In the case of this application, a good README document should allow a user with limited knowledge of ROS2, Gazebo and coding to complete an agility robot simulation successfully. It can be considered a set of instructions for a simulation.
The first section of the README document states the minimum required versions of ROS2, Gazebo, Python and C++ that the simulation had been developed on and are therefore needed for the project to operate. The following lines explain a step-by-step procedure that helps users start and control a simulation. First the Github repository should be cloned on the users system, this will include a directory containing all the CAD mesh files and a directory of the project package itself. After that the absolute paths in the robot's SDF file and world file should be updated. This is because a new system will have the project saved in a different location than the originally developed project. After this had been done the package needs to be built using 'Colcon build' such that the project has access to the correct code libraries that come with ROS2 and Gazebo. Next the project needs to be sourced in the terminal and the launch file needs to be executed to start the simulation. Finally in a separate terminal the robot can be controlled with the keyboard.
The corresponding terminal commands have also been included at each step which means they can be quickly copied and pasted into the command line which makes repeating simulations are easy, quick and efficient.
Another aspect to consider is that this project and its packages will be uploaded onto Github to allow accessibility for the university and constant updates on the project. All Github repositories are required to contain a README document.

This concludes the progress achieved for the first year of development on the agility robot's car-in-the-loop simulation.

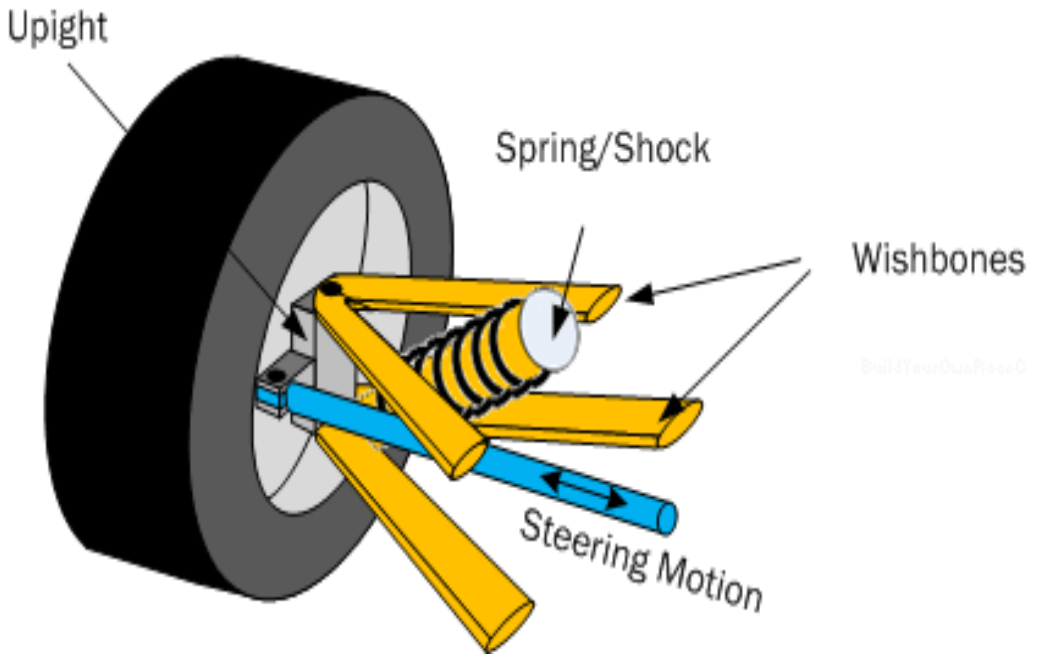
4. Project Evaluation

This section of the report is dedicated to critically evaluate the overall success of this academic year of development. The main aim of an in-the-loop simulation is to create a realistic recreation and therefore the success of the project is mainly measured by the accuracy of the simulated robot in comparison to the real agility robot. Other measures of project success are the justification of the design choices, the total functionality of the code and the amount learnt during the year.

To judge the accuracy of the robot the CAD models, parameters and joint configurations will be evaluated and discussed.
The CAD mesh files had a factor in creating an accurate robot as they visually created the robot and created the solid volume the robot used up in the simulation. The mesh files were created to a good level of accuracy as they were made to the correct specifications, when dimensions were given. However, some technical drawings and dimensions weren't available which led to some dimensions being estimated to gain the general shape. For example, the chassis bumpers didn't have any dimensions and the wheel width wasn't specified. Additionally, a technical drawing of the chassis didn't exist, but only critical values were given. Consequently, CAD models were created to capture the critical dimensions given such as wheelbase and wheel track but the non-critical dimensions and were not visually accurate. The mesh files fulfil their intended purposes well and the collision body looks very similar to the agility robot even if some non-critical parts weren't given. Consequently, this aspect of the robot is successful.

Arguably the most important aspect of the robot lies with the parameters given to each link and joint. Generally, this section of robot creation has massive potential for improvement. Most values given to the links and joints were based on informed estimations rather than from quantitative facts. This is because the agility robot had already been created by the time the simulation started development whereas if the robot was disassembled, each part could be assessed for more accurate parameters. This would be done by weighing each part individually to get an accurate mass for each link. Additionally, if the robot could be disassembled the spring-damper for the robot suspension could be put into a test rig and investigated to gain the correct values of spring stiffness and damping ratio. This would significantly increase the suspensions accuracy and therefore increase the quality of the entire simulation. Another link that could see substantial improvement to its nature are the tyres. Although the simulator has some limitations in mimicking realistic tyres it would've been preferable to physically test the agility robot's tyres to gain experimental data for its coefficient of friction and coefficient of restitution. This experimental data could then be implemented into the tyre parameters to make them authentic.
Considering these points and the fact that most parameters were based on estimates, it demonstrates there is space for parameter improvement. As a result, it's fair to suggest the parameter selection was only a mild success. However, this was only caused by a factor out of the report writers control. The robot couldn't be broken down into its components as it had already been created with complex electronics and disassembling the robot would be counterproductive. Therefore, the parameters selected were appropriate in consideration of this obstacle.

Another point that needs evaluation is the configuration of joints within the robot. The suspension design had some simplifications that differed from the real agility robot. The simplifications are that the spring-damper shock absorber component wasn't added to the simulation but rather its dynamics were included by adding its parameters to a joint. Another simplification meant that the steering system was solely defined by two joints whereas the real agility robot followed a conventional steering system, with the wheels connected to a steering column controlled by a motor. This configuration is similar to the one shown in figure 4.1.



(Suspension-Basics, 2015)

Figure 4.1 – Diagram showing the configuration of double-wishbone suspension with a steering mechanism.

The simulated robot didn't contain the exact suspension and steering configuration as the agility robot and opted for a simpler design. As a result, the joint configuration may appear unsuccessful as it doesn't accurately represent the real robot. However, from the nature of simulations, the steering bar and shock absorber don't need to be included as these aspects are controlled with topics and parameters instead. Even though not all components are used, the overall working dynamics and functions are the same and create movements true to the real robot. As a result, the joint configuration can be deemed a success as they produce accurate dynamics and have also saved time by opting for a simpler configuration that would requires less joints and CAD models.

The next measure of success is the working function of the simulation and if the code worked without errors. Overall, the structure of the package followed the standard format which meant the project's packages are suitable for a Github repository. Furthermore, all files and sections of code has been described for clarity and readability. The Cmakelist compiled the package successfully. Additionally, the launch file runs the simulation with the robot, environment, required packages and plugins which makes the setup easy and time efficient. Consequently, the code is working as intended and suggests the technical side of the simulation has been successful. However, one downside of the package is that when it's downloaded to another computer system the absolute paths in the code needs to be updated. This is a problem because there are several absolute paths to mesh files located within the SDF files and absolute paths in the world file. This may be inconvenient to change at first and suggests a point to improve upon. Additionally, the teleop_keyboard node needs to be run from the terminal which adds an extra step to start, when actually this node could be added to launch file to automatically start in parallel with a simulation. In total, the code works effectively without errors which is the main aim of the project, even if it could become more convenient in some areas. Therefore, this part of the project can be regarded as an achievement.

The final measure of success required a literature review around the subject. Robotics and in-the-loop simulations require extensive research as there are many different essential aspects/files to consider that make a successful simulation. For example, the robot in question needed to be known in depth to re-create it accurately. Also, a high level of complex coding was needed in order to produce the technical side of the simulation. Nevertheless, the research gathered from the duration of the project can be used to gain a respectable understanding of in-the-loop simulations. This report explains why a simulation may be needed and how to create such simulations from scratch. The required files have been explained in detail and the agility robot package presents a working example that incorporates all the learnt knowledge into an effective simulation tool. Thus, it can be said with confidence that this document would be beneficial for anyone who wants to learn how to create a robotic in-the-loop simulation. Overall, these points show that a very useful, complex skill has been developed and that the knowledge gained can be applied to make new robotic simulations in the future. As the result, the academic, learning side of the project could be deemed a success.

Suggesting the project was a success is relative and subjective but taking all these points into account there is no doubt the agility robot simulation will become a useful tool for the robot's development and that during this academic year a wider breath of knowledge has been acquired. Therefore, it was worth creating.
However, this doesn't mean the current state of the project is perfect which leads onto the intended future improvements of the simulation which would make the simulation more useful and accurate.

5. Future Improvements

Unlike other final year projects, this project is intended to be continued in tandem with further development of the agility robot. As a result, the simulation will become extremely advanced and accurate over time. Eventually it can help test driving characteristics of the agility robot and speed up its development with precise simulation testing. Though this is not possible if the simulated robot is behind in development in comparison to the real robot. This section of the report suggests the steps needed to develop the simulation to an equal level to that of the real robot.

As previously mentioned, some parameters lack accuracy. Mainly the mass of each link and the spring-damper joint values. This isn't a major issue but if the agility robot could be broken down into its individual components it would be beneficial to measure the components mass and the suspension spring and damping values by experimental method. There is a high chance this opportunity will not present itself and means the parameters currently selected are suitable.

Most of the remaining functionality can be implemented by use of more plugins. The first plugin implemented should be a tweaked version of the 'ackermann_drive' plugin which is part of 'gazebo_ros' package. This plugin should contain most of the original code from ackerman_drive plugin file but should allow for the exclusion of a steering wheel joint. This means editing the filename 'gazebo_ros_ackermann_drive.cpp' such that the steering wheel functionality is either removed or the plugin is altered such that it creates the intended four-wheel drive, front Ackermann steering dynamic while allowing the steering wheel joint to be an empty parameter. After this plugin has been implemented the robot will be massively improved and the motion would be a near to perfect recreation of the agility robot. Additionally, the method of controlling the robot should subscribe to the plugin automatically. The current configuration employs keyboard inputs for the robot control. This should be added to the launch file such that it automatically opens for the user and allows the control of the Ackermann drive plugin.

The robot also requires a camera that provides a live video feed from the robot's point of view. This is an essential part because, in reality, this is the only form of perception the user has when driving the robot. Consequently, viewing the robot from a bird's eye view in the gazebo environment is not realistic and adding a camera would greatly increase the user feedback accuracy. Adding a camera can be achieved by implementing a camera link in the correct location of the real camera. The camera body is already included by the chassis' mesh file, so the correct location of the camera is identifiable.

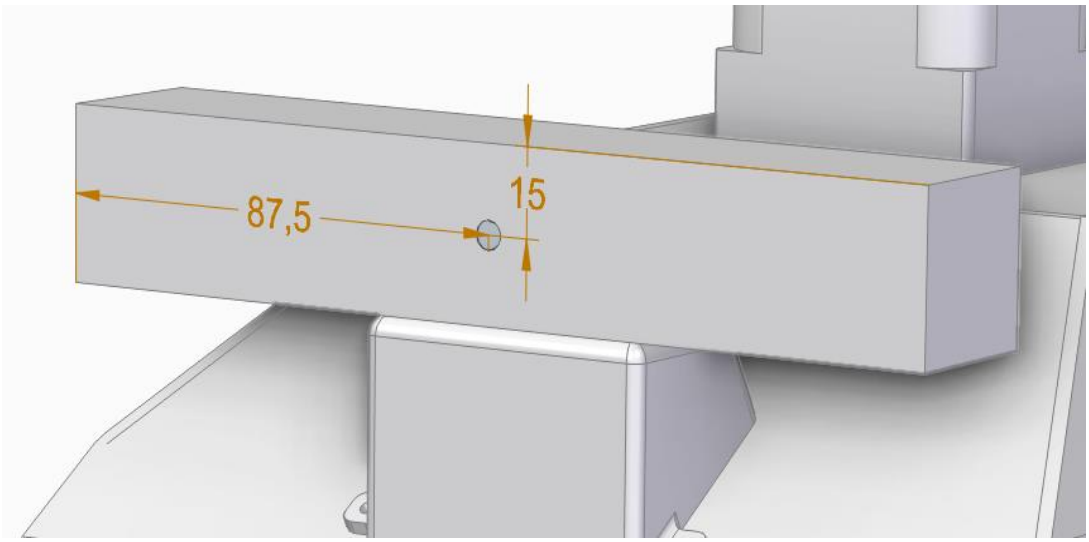


Figure 5.1 – Model showing suggested location of camera link.

The location of the camera can be placed in the middle of the camera body, as shown in figure 5.1. A camera link has depth capability with only one lens, whereas the real agility robot needs two lenses for depth information. Consequently, the position of the camera will be different in the simulation in comparison to the real camera location.
After that the 'gazeob_ros_camera' plugin should be implemented in the robot' SDF file to give the camera link its intended function. Finally, the images sent via the camera's topic should be connected to a separate window to build up the camera's live feed. This can be achieved using a software called RVIZ. RVIZ can create a screen that subscribes to the camera topics and graphically displays the images at a certain update rate, effectively creating a live feed to the simulated world that is separate from Gazebo's GUI.

The agility robot also contains a lidar sensor. This will be used to detect objects 360 degrees around the robot and also gathers information on object distances. Consequently, the gathered information from a lidar sensor is very useful for navigation and collision avoidance for autonomous algorithms. At this point in time the agility robot doesn't make use of its lidar sensor as its controlled manually, therefore using the front camera as feedback to the driver is more advantageous. However, once the robot implements its autonomous driving the lidar will be a great source of information and feedback. As a result, it needs to be implemented into the simulation. This is done by creating a link that represents the sensor. After that, a ROS2 sensor, of type 'ray', needs to be added. Finally, a plugin of name 'libgazebo_ros_ray_sensor.so' should be attached to the previously created lidar link inside the SDF file. This design will create a laser scan topic which can be subscribed to by the autonomous driving algorithms. As the shape of the lidar sensor has already been included in the simulation, the lidar link could be unnecessary and the sensor could just be positioned in the correct position as suggested in figure 5.2.

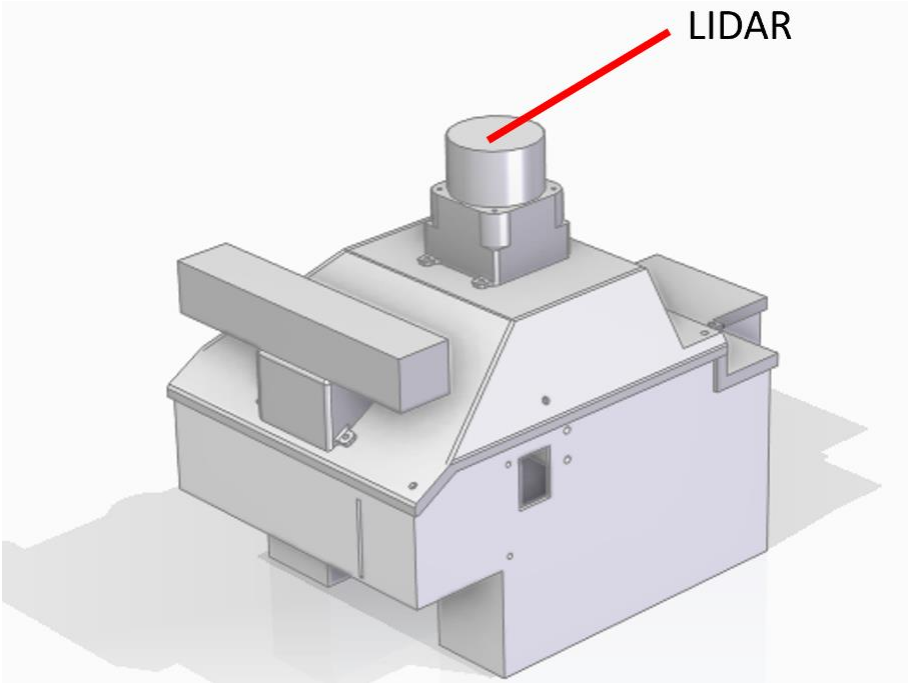


Figure 5.2 – Model showing position of the LIDAR sensor.

One aspect of the robot that could be improved is the inner configuration of components inside the control box. Currently the chassis and control box have a combined mass that includes all the components inside. This is an assumption which may reduce the accuracy. If the chassis mass was changed to exclude the inner components the robots centre of mass and centre of mass position could change. This is because the components inside make up most the mass inside the chassis. Consequently, adding these individual components with their masses in the correct location would change the vehicles mass distribution and alter the dynamics. The location of centre of mass has large effect on the vertical tyre load which changes the cornering and breaking characteristics. Evidently, adding each component (motherboards, battery, etc.) separately would increase the mass distribution accuracy and therefore improve the dynamics of the agility robot in the simulation. This can be implemented by mapping out the inside of the control box and adding links for each component inside. The joints between these links should be rigidly fixed to the control box housing.

Another improvement that could be made to the simulation is adding a WIFI connection to the robot. As the agility robot requires a WIFI connection, simulating the areas where the robot is in the range of the WIFI router's signal and when the connection is lost will be a useful addition to the simulation.

The addition of a battery life may also be useful for testing the agility robot. If there was a node which measured power level in a battery which lowered when the robot moved it would be a decent indicator of the driving range and how aggressive the robot could drive within its battery time.

Finally, during the stage where the autonomous driving algorithms are being developed the code that makes up these algorithms should be created as ROS2 nodes. The nodes would then be implemented into the simulation and tested. There may be a large number of algorithms that work together to create the total decision making and dynamic responses in reaction to the robots surrounding. This may make implementing it into the real robot a difficult task. Consequently, as development is occurring each algorithm's function can be

added to the simulation individually, or with other functions if they work correctly together and give the desired outcome. As a result, each node can be changed accordingly without having to add the hardware to the real robot or test the real robot with an algorithm that has potential to cause an accident. This would be achieved by coding custom plugins that have similar code to that of the autonomous algorithms. The code structure between the simulation plugins and the real-life algorithms would be slightly different as the decisions would change the data on the topics and joints rather than create electrical power in the robots hardware, such as motors.
This may take a significant amount of time and extra research to achieve as custom plugins weren't covered in the scope of this year. But once the methods had been found the simulation can be constantly updated with the robot changed to help test every individual change.

6. Conclusion

To conclude, the simulation is at a good stage given the time frame. Although the dynamics need to be improved to generate valuable simulations. The given suggestions for further improvements should create a very accurate robot that acts the same as the agility robot.

The first aim of the project was to create an effective literature review of in-the-loop simulation. During this year of development, a large amount of research was generated and reported. From the report a lot can be learnt about in-the-loop simulations and the nature of ROS2 and Gazebo. It explains the benefit and working principles of simulations in the context of the agility robot. Furthermore, each individual aspect required to make a robotics package and robotics simulations have been described and supported by the agility robot package as an example. Reading this document gives a larger understanding of launch files, robot description files, mesh files, sensor, plugins and more which are essential for working in ROS2 and gazebo. Considering this, the first aim of the project has been achieved and the research generated would be beneficial for anyone interested in making a robot simulation and is recommend for anyone developing the agility robot project in the future.

The second aim required a robotic simulation of the agility robot. This has been started but requires some work to become the valuable testing tool that the project is intended to become. Nevertheless, the current progress on the simulation is good quality and has created a strong base for new developers to build upon. If the suggested improvements are implemented into the project, there is a high chance that the simulation will become very accurate and valuable within one more year of work. As a result, it will aid the development of the agility robot in becoming fully autonomous and quicken the testing time of autonomous driving algorithms.

Overall, it will be interesting to see the improvements and progress made in the following years, but this particular year has been a great learning experience.

7. References

(Automaticaddison, 2021) "*Organizing Files and Folders inside a ROS 2 Package*." Automaticaddison.com, 4 Nov. 2021, automaticaddison.com/organizing-files-and-folders-inside-a-ros-2-package/#:~:text=In%20a%20ROS%202%20project,to%20be%20inside%20a%20package.. Accessed 11 Apr. 2022.

(AutoRally, 2020) "Autorally/README.md at Melodic-Devel · AutoRally/Autorally." *GitHub*, 13 July 2020, github.com/AutoRally/autorally/blob/melodic-devel/README.md. Accessed 4 May 2022.

(Contruct- Sensors, 2020) "*A List of ROS2 Supported Sensors for Robots - the Construct*." *The Construct*, 26 Oct. 2020, www.theconstructsim.com/list-ros2-supported-sensors-for-robots/. Accessed 27 Apr. 2022.

(Lifewire-DAE, 2022). "*What Is a DAE File and How Do You Open One*?" *Lifewire*, 2022, www.lifewire.com/dae-file-2620544. Accessed 18 Apr. 2022.

(Lifewire-XML, 2022) "*What Is an XML File and How Do You Open One?" Lifewire*, 2021, www.lifewire.com/what-is-an-xml-file-2622560. Accessed 18 Apr. 2022.

(Linux, 2022) "*The Linux Command Line for Beginners | Ubuntu*." *Ubuntu*, 20 ubuntu.com/tutorials/command-line-for-beginners#9-conclusion. Accessed 12 Apr. 2022.

(OSRF-Joints, 2020). "*SDFormat Specification.*" *Sdformat.org*, 2020, sdformat.org/spec?elem=joint. Accessed 12 Apr. 2022.

(OSRF-physics, 2014) "*Gazebo : Blog : Gazebo Supports Four Physics Engines*." *Gazebosim.org*, 2014, gazebosim.org/blog/four_physics. Accessed 7 Apr. 2022.

(OSRF-Plugins101, 2012). "*Gazebo : Tutorial : Plugins 101*." *Gazebosim.org*, 2012, gazebosim.org/tutorials/?tut=plugins_hello_world. Accessed 14 Apr. 2022.

(OSRF-URDF, 2014). "*Gazebo : Tutorial : URDF in Gazebo.*" *Gazebosim.org*, 2014, gazebosim.org/tutorials/?tut=ros_urdf. Accessed 12 Apr. 2022.

(OSRF-stdpackage, 2014). "*Gazebo : Tutorial : ROS 2 Overview*." *Gazebosim.org*, 2014, gazebosim.org/tutorials?tut=ros2_overview. Accessed 14 Apr. 2022.

(Robotogeddon, 2020) "*ROS2 Package Creation Architecture.*" *YouTube*, 23 July 2020, www.youtube.com/watch?v=Y98R2ysIoE4&ab_channel=Robotogeddon. Accessed 11 Apr. 2022.

(Ros development, 2020) "*ROS1 vs ROS2, Practical Overview for ROS Developers - the Robotics Back-End*." *The Robotics Back-End*, 25 May 2020, roboticsbackend.com/ros1-vs-ros2-practical-overview/#:~:text=ROS1%2C%20initially%20created%20in%202007,the%20open%20source%20robotics%20community.. Accessed 14 Apr. 2022.

(ROS Wiki- gazebo_ros, 2015) "*Gazebo_ros_pkgs - ROS Wiki*." *Ros.org*, 2015, wiki.ros.org/gazebo_ros_pkgs#Overview. Accessed 14 Apr. 2022.

(ROS Wiki- Sensors, 2022) "*Sensors - ROS Wiki*." *Ros.org*, 2022, wiki.ros.org/Sensors#Portals. Accessed 27 Apr. 2022.

(ROS1 ROS2 changes, 2015) "*Changes between ROS 1 and ROS 2.*" *Ros2.org*, 2015, design.ros2.org/articles/changes.html. Accessed 14 Apr. 2022.

(ROS2Docs, 2021) "*ROS 2 Documentation — ROS 2 Documentation: Foxy Documentation*." *Ros.org*, 2021, docs.ros.org/en/foxy/index.html. Accessed 11 Apr. 2022.

(ROS2DocsLaunch, 2021) "*Creating a Launch File — ROS 2 Documentation: Foxy Documentation.*" *Ros.org*, 2021, docs.ros.org/en/foxy/Tutorials/Launch/Creating-Launch-Files.html. Accessed 12 Apr. 2022.

(Suspension-Basics, 2015) "*Car Suspension Basics, How-to & Design Tips!*" *Build Your Own Race Car!*, 2015, www.buildyourownracecar.com/race-car-suspension-basics-and-design/4/. Accessed 26 Apr. 2022.

(Wikipedia – Sensors, 2022) Wikipedia Contributors. "*Robotic Sensors.*" *Wikipedia*, Wikimedia Foundation, 7 Feb. 2022, en.wikipedia.org/wiki/Robotic_sensors. Accessed 27 Apr. 2022.

8. Appendix

	Link Name	Description
1	Chassis	Includes the complete description of the robot's chassis and control housing. It also includes the shape of the lidar and camera sensor.
2	FLBwing	The bottom control arm in the front, left double wishbone suspension .
3	FLTwing	The top control arm in the front, left double wishbone suspension.
4	FLspindle	The front left spindle in the double wishbone suspension which the wheel rotates around and allows the wheel to steer.
5	FLwheel	The wheel and tyre on the front left side
6	FRBwing	The bottom control arm in the front, right double wishbone suspension .
7	FRTwing	The top control arm in the front, right double wishbone suspension.
8	FRspindle	The front right spindle in the double wishbone suspension which the wheel rotates around and allows the wheel to steer.
9	FRwheel	The wheel and tyre on the front right side
10	BLBwing	The bottom control arm in the back, left double wishbone suspension.
11	BLTwing	The top control arm in the back, left double wishbone suspension.
12	BLspindle	The back left spindle in the double wishbone suspension which the wheel rotates around and allows the wheel to steer.
13	BLwheel	The wheel and tyre on the back, left side
14	BRBwing	The bottom control arm in the back, right double wishbone suspension .
15	BRTwing	The top control arm in the back, right double wishbone suspension.
16	BRspindle	The back, right spindle in the double wishbone suspension which the wheel rotates around and allows the wheel to steer.
17	BRwheel	The wheel and tyre on the back, right side.

	Joint Name	Joint Type	Parent link	Child link
1	chassis_FLBwing_joint	Revolute	Chassis	FLBwing
2	chassis_FLTwing_joint	Revolute	Chassis	FLTwing
3	spindleFLB_joint	Ball	FLBwing	FLspindle
4	spindleFLT_joint	Ball	FLTwing	FLspindle
5	FL_steer	Revolute	FLBwing	FLspindle
6	wheelFL_joint	Revolute	FLspindle	FLwheel
7	chassis_FRBwing_joint	Revolute	Chassis	FRBwing
8	chassis_FRTwing_joint	Revolute	Chassis	FRTwing
9	spindleFRB_joint	Ball	FRBwing	FRspindle
10	spindleFRT_joint	Ball	FRTwing	FRspindle
11	FR_steer	Revolute	FRBwing	FRspindle
12	wheelFR_joint	Revolute	FRspindle	FRwheel
13	chassis_BLBwing_joint	Revolute	Chassis	BLBwing
14	chassis_BLTwing_joint	Revolute	Chassis	BLTwing
15	spindleBLB_joint	Ball	BLBwing	BLspindle
16	spindleBLT_joint	Ball	BLTwing	BLspindle
17	wheelBL_joint	Revolute	BLspindle	BLwheel
18	chassis_BRBwing_joint	Revolute	Chassis	BRBwing
19	chassis_BRTwing_joint	Revolute	Chassis	BRTwing
20	spindleBRB_joint	Ball	BRBwing	BRspindle
21	spindleBRT_joint	Ball	BRTwing	BRspindle
22	wheelBR_joint	Revolute	BRspindle	BRwheel