School of Engineering, Discipline of Mechanical Engineering and Mechatronics, pp. 2–44

QCar FYP Simulation Guide

August 2023

Jacob Davison, Elliott Shore [1]

[1] *The University of Newcastle, Callaghan, NSW 2308, AUSTRALIA*

Contents

1 Linux Virtual Machine Installation and Setup

To create and run the ROS applications required within the QCar platform and Gazebo simulation, a Linux operating system is required. To emulate a Linux system on your current Windows machine requires the use of a Virtual Machine.

1.1 Required Software

Download the required Virtual Machine application VMware Workstation Player and an Ubuntu Linux Distribution ISO file. Ubuntu Mate is the recomended distribution as it has been tested and used with Gazebo and ROS. When downloading Ubuntu ensure a 64-bit Desktop Image is chosen and save the respective ISO file to a location you will remember.

- VM Install Link: https://www.vmware.com/au/products/workstation-player.html.
- Ubuntu Mate: https://cdimage.ubuntu.com/ubuntu-mate/releases/focal/release/

1.2 Setup Virtual Machine

The following instructions were created by following YouTube tutorials provided by the channel Articulated Robotics, who creates lots of useful content on how to use ROS, Gazebo and other robotics related projects.

- Articulated Robotics Channel: https://www.youtube.com/@ArticulatedRobotics/videos
- How to setup VM: https://www.youtube.com/watch?v=laWn7_cj434&t=290s
- How to setup ROS: https://www.youtube.com/watch?v=uWzOk0nkTcI&t=169s

1.2.1 Setup Steps

1. Ensure you have both VMware Workstation Player and an Ubuntu ISO file installed.
2. Open VMware workstation player and select create a new virtual machine.
3. Select install operating system later and select next.

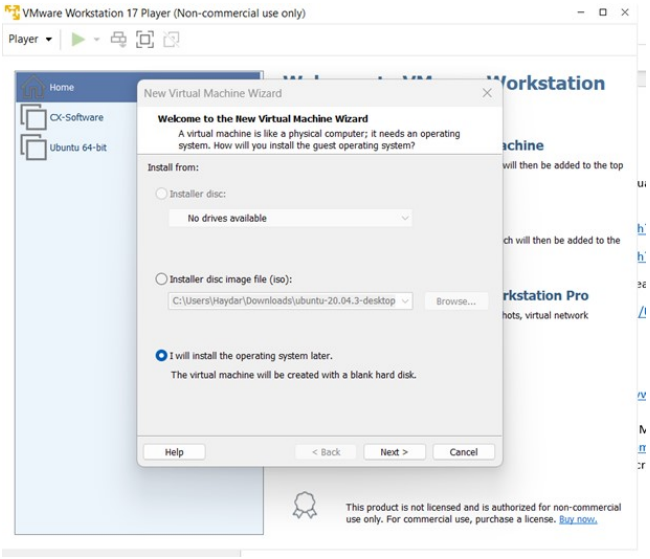


Figure 1: Select Install Operating System Later.

4. Ensure the guest operating system is Linux and version is Ubuntu 64-bit.

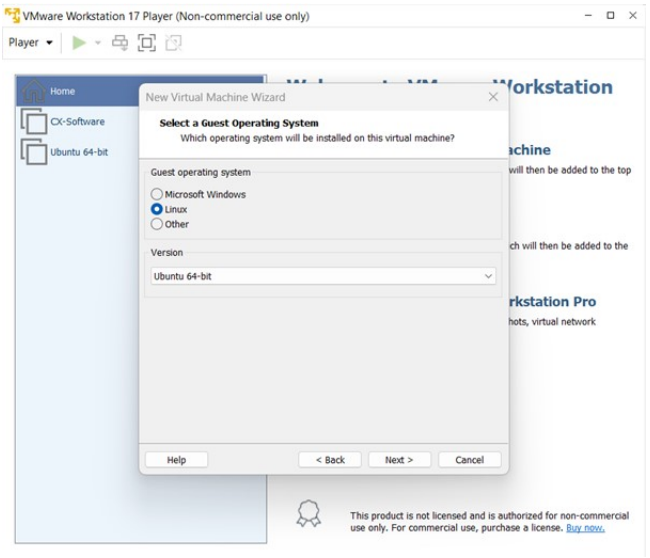


Figure 2: Select Linux Ubuntu 64-bit.

5. Name your project and select a folder to store it in and hit next.

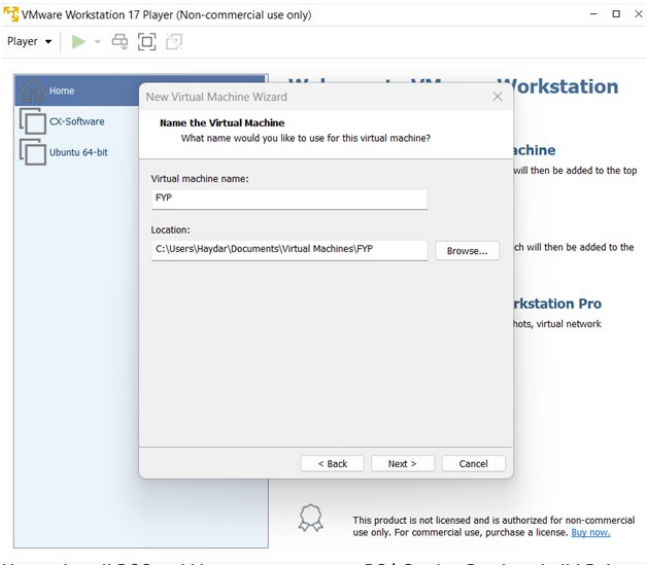


Figure 3: Select VM Location And Name.

6. Select store in single file and increase the disk space as the simulation environment can grow quite large over time. This can also be changed manually later.

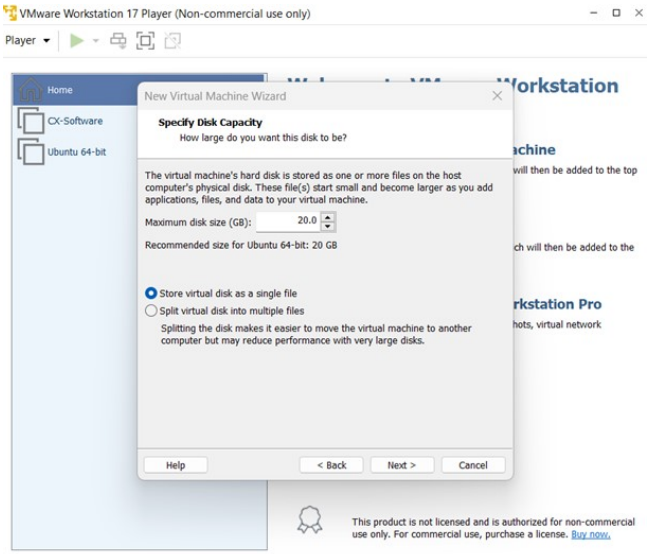


Figure 4: Select Store in Single Disk and Increase Disk Size.

7. Hit finish on the virtual machine setup. **BEFORE RUNNING IT**, go to virtual machine settings → CV/DVD (SATA) and select use ISO image file. Find your Ubuntu ISO file and select it. Also, navigate into virtual machine settings → display and turn off accelerate 3D graphics. This setting causes issues in gazebo and prevents models displaying in simulation.

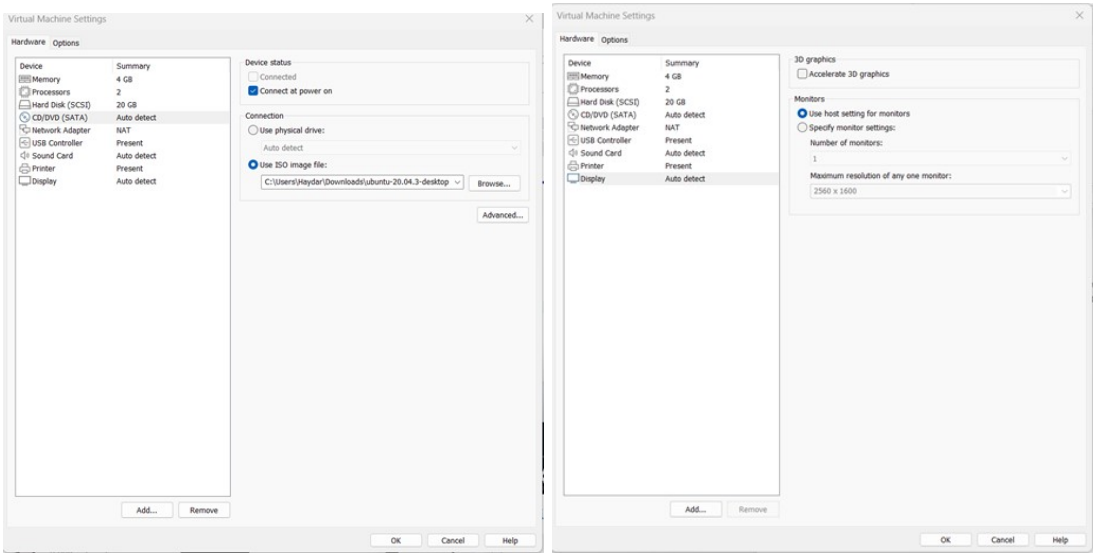


Figure 5: Left: Selecting ISO Image File within CV/DVD (SATA), Right: Turn Off Accelerate 3D Graphics in Display

Tip for the virtual machine, the memory and disk size of the machine can be edited once shutdown in the virtual machine settings:

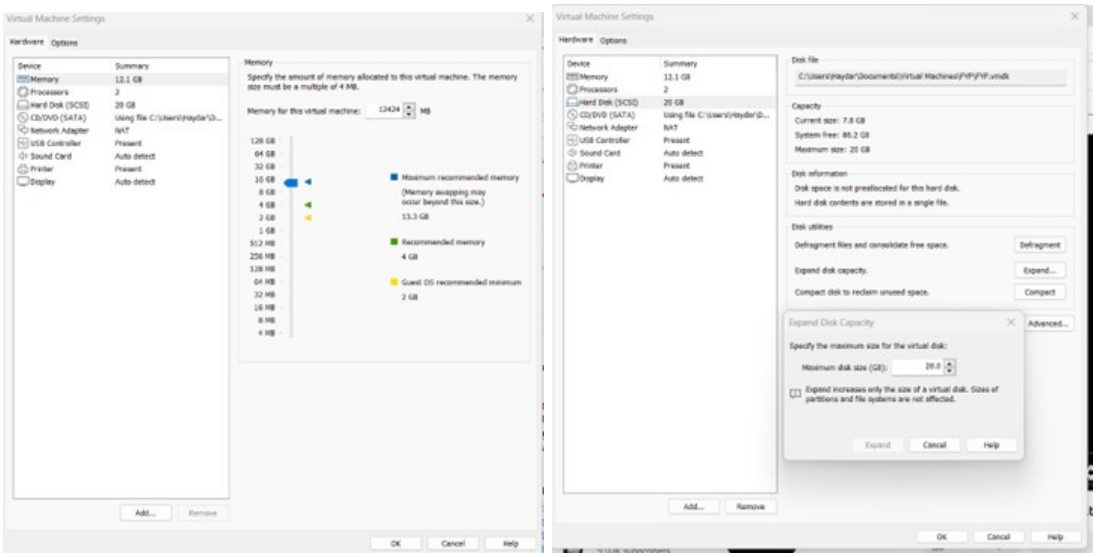


Figure 6: Left: Increase Memory of VM, Right: Increase Maximum Disk Size of VM

8. Exit settings and play the virtual machine.

9. Once in the virtual machine, wait for the Install screen to appear and select install Ubuntu.

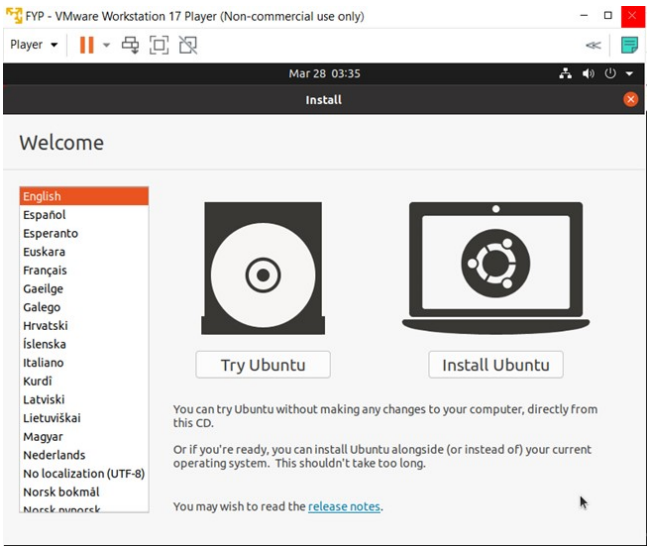


Figure 7: Select Install Ubuntu.

10. Select ok for all steps until Ubuntu asks to create a username and password. Fill in required info.

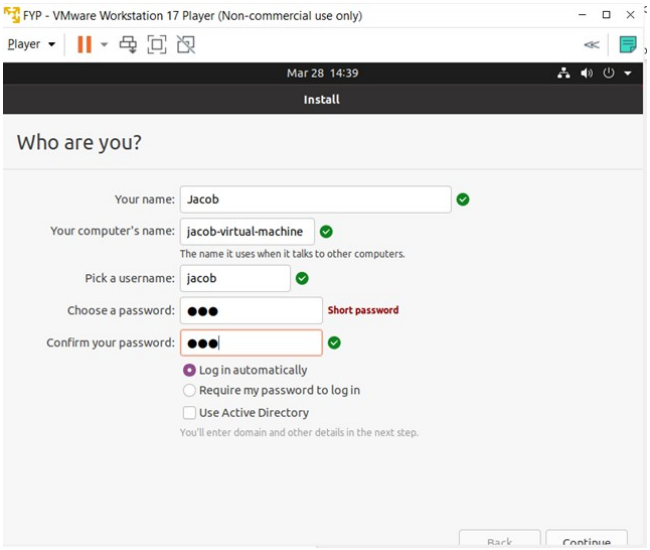


Figure 8: Create VM Username and Password.

11. The Virtual Machine setup is complete. Follow the next section to setup ROS within the VM.

1.3 Setup ROS within VM

To run the QCar within a Gazebo simulation requires ROS (Robot Operating System), which is a framework and set of tools that act as an operating system for use within robotics. The ROS distribution used within this project will be ROS Noetic as it is the most current and supported ROS1 distribution. The QCar utilises a ROS1 distribution and thus ROS2 programs will not operate on the QCar. The install instructions for ROS Noetic were created following the instructions outlined on the ROS website.:

- ROS Noetic Install Instructions: http://wiki.ros.org/noetic/Installation/Ubuntu

1.3.1 Setup Steps

1. Within the VM, open up a terminal
2. Setup the ROS sources by running the code below. If copying the commands, delete and re-type the ' characters.

Listing 1: Setup ROS Noetic source.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. Setup the required keys by running the code below.

```
root@ubuntu:~$ sudo apt install curl
root@ubuntu:~$
    curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc  sudo
    apt-key add -
```

4. Update the Linux package list by running the code below.

```
root@ubuntu:~$ sudo apt update
```

5. Complete a full desktop install of ROS Noetic to ensure that the Gazebo simulator and tools are also installed, using the code below.

```
root@ubuntu:~$ sudo apt install ros-noetic-desktop-full
```

6. Before using any ROS tools, the environment setup script must be run. This can be done by running the code below.

```
root@ubuntu:~$ source /opt/ros/noetic/setup.bash
```

However, to prevent the need to run this command everytime a new terminal is opened up, the following command can be run that adds this setup command to the startup script for the terminal.

```
root@ubuntu:~$ echo "source␣/opt/ros/noetic/setup.bash" >> ~/.bashrc
root@ubuntu:~$ source ~/.bashrc
```

7. To prevent dependency issues the following command should then be run.

```
root@ubuntu:~$ sudo apt install python3-rosdep python3-rosinstall
    python3-rosinstall-generator python3-wstool build-essential
```

8. Install rosdep, a ROS tool useful for easily installing system dependencies.

```
root@ubuntu:~$ sudo apt install python3-rosdep
```

Then use the following to initialise rosdep.

```
root@ubuntu:~$ sudo rosdep init
root@ubuntu:~$ rosdep update
```

9. Test the install running gazebo, an empty world should launch.

```
root@ubuntu:~$ gazebo
```

10. Further install dependencies required for the project.

```
root@ubuntu:~$ sudo apt git
root@ubuntu:~$ sudo apt install python3-pip
root@ubuntu:~$ python3 -m pip install scipy
```

1.4 Clone and Setup workspace from GITHUB

For this project, a premade ROS workspace was created that allows for simulation of the QCar model within a Gazebo simulation environment. The QCar model files used within this project are provided on the Quanser website in the QCar research papers section.

- Quanser QCar Website https://www.quanser.com/products/qcar/

1.4.1 Setup Steps

1. In a new terminal, create a new empty directory to clone the workspace into and move into that directory using the following commands, changing the workspace name to one of your choice.

```
root@ubuntu:~$ mkdir catkin_ws
root@ubuntu:~$ cd catkin_ws
```

2. Clone the workspace into the current folder using the git clone command shown below. The full repository can be viewed at https://github.com/jacobdavo/UON-QCAR-BASE.

```
root@ubuntu:~/catkin_ws$
    git clone https://github.com/jacobdavo/UON-QCAR-BASE.git
```

3. After cloning the workspace, it must be built using the following catkin command. After every build the workspace must be setup before running any code. The following source command sets up the current terminal to use the files provided within the workspace.

```
root@ubuntu:~/catkin_ws$ catkin_make
root@ubuntu:~/catkin_ws$ source devel/setup.bash
```

4. The workspace is fully cloned and setup for use!

2 Workspace Setup for ROS and Gazebo

NOTE: Prior to running any ROS related commands in each new bash terminal, the installation of ROS on your personable machine must be sourced. This can be done using the below command, replacing 'noetic' with your installed distribution of ROS.

```
root@ubuntu:~$ source /opt/noetic/ros/setup.bash
```

In a new bash terminal create a new directory `workspace/src` and navigate to the directory root file.

```
root@ubuntu:~$ mkdir -p ~catkin_ws/src
root@ubuntu:~$ cd catkin_ws
```

The workspace can then be initialised using the `catkin_make` command from within the workspace root folder. This builds all essential catkin files, adding a build and devel folder to the root workspace folder as well as a CMakeLists.txt file to the src folder.

```
root@ubuntu:~/catkin_ws$ catkin_make
```

Following the successful creation of the workspace, a catkin package can be created from within the src folder using the command `catkin_create_pkg`.

```
root@ubuntu:~/catkin_ws/src$ catkin_create_pkg catkin_package roscpp rospy std_msgs
```

The above command use, creates a catkin package called `catkin_package` and specifies dependencies to the `roscpp`, `rospy` and `std_msgs` libraries. Following this, the workspace must be built once more using `catkin_make` from the workspace root folder.

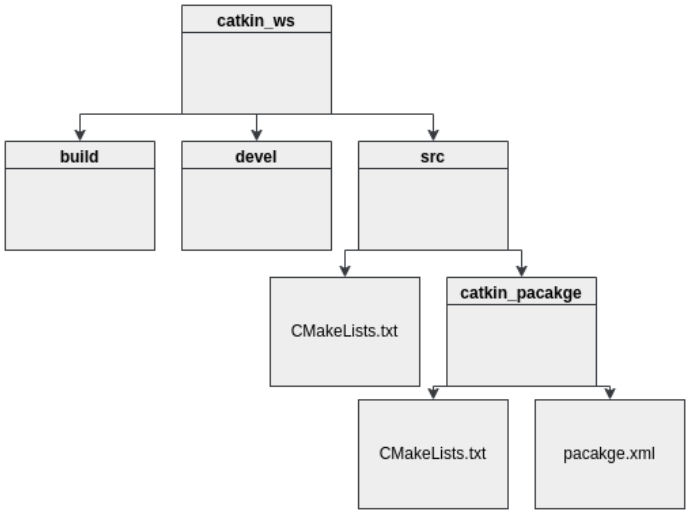


Figure 9: Catkin workspace folder structure.

3 ROS Basics

ROS provides an operating system to the robot both in the physical QCar and in simulation. ROS provides a system of communication between 'Nodes' which are executable programs made in Python, C++, and other languages that communicate between each other using ROS topics.

3.1 ROS Launch files

With reference to the launch file *UON-QCAR-BASE/src/qcar_gazebo/launch/Track1.launch*, this section will outline how launch files work. Launch files are used by the `roslaunch` command to start multiple nodes within a single command, as well as other simulation parameters and arguments. In addition a launch file will automatically start an instance of *roscore* (provided one is not already running) which is necessary for all ROS related programs to communicate.

Launch files are written in the XML programming language and are contained within the `<launch>` and `<\launch>` tags.

Listing 2: Launch files arguments.

```
    <!-- Set initial robot pose -->
    <arg name="x" default="-4.3"/>
    <arg name="y" default="3"/>
    <arg name="z" default="0"/>
    <arg name="roll" default="0"/>
    <arg name="pitch" default="0"/>
    <arg name="yaw" default="-3.14"/>
```

Listing 2 illustrates the use of the `<arg>` tag which allows for the definition of local reusable parameters from within the launch file. Specifically, the above example defines the initial pose of the qcar in space.

Listing 3: Launch file parameters.

```
1  <!-- Send the URDF file to the ROS parameter server -->
2  <param name="robot_description" command="$(find xacro)/xacro $(find qcar_gazebo)/urdf/qcar_model.xacro" />
```

Listing 3 shows the use of the `<param>` tag. This is used to define a parameter on the ROS parameter server an is available across all currently active ROS nodes. For example; line 2 sets a parameter 'robot_description' that imports the file *qcar_model.xacro* using the command attribute.

Below, Listing 4 shows the `<rosparam>` tag. This tag is used in a similar way to the `<param>` tag, except makes use of a `.yaml` file to load or dump parameters to the ROS parameter server. The below example therefore loads the parameters specified in the *qcar_controlplugin.yaml* file.

Listing 4: Launch file ROS parameters.

```
1  <rosparam file="$(find qcar_controlplugin)/config/qcar_controlplugin.yaml" command="load"/>
```

Launch files also make use of the `<include>` tag which can include other files giving the launch access to the contents of said file. Below in Listing 5 a `<launch>` tag is used to include another launch file *empty_world.launch* which launches a basic Gazebo simulation world. Arguments are then passed into the world launch file which specify the contents of the world being launched.

Listing 5: Launch file include.

```
1  <!-- Launch the Gazebo World -->
2  <include file="$(find gazebo_ros)/launch/empty_world.launch">
3     <arg name="world_name" value="$(find qcar_gazebo)/worlds/Track1.world"/>
4     <arg name="gui" value="true"/>
5     <arg name="verbose" value="true" />
6  </include>
```

Finally, launch files use the `<node>` tag to launch predefined ROS nodes. This can be seen below in Listing 6, where the `node` tag is used to specify the node *joint_state_publisher*. The attribute *pkg* specifies the package in which the node can be found, *type* specifies the name of node and *name* is an optional attribute which can be used to overwrite the nodes name.

Listing 6: Launch file nodes.

```
<!-- Setup the joint state publisher to publisher the state of the robot -->
<node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
</node>
```

3.2 ROS Nodes

ROS Nodes are the executable files created to perform the specified tasks of the robot (Control, Guidance, etc). In python, ROS Nodes can be initialised using the following code which can be found in *UON-QCAR-BASE/src/qcar_guidance/example_guidance_node.py*.

Listing 7: Initialise Python Nodes.

```
import rospy            #imports the ROS Python library
if __name__ == '__main__':
        rospy.init_node('example_guidance_node') #Initalises the Node
```

Nodes can be launched two ways, using a launch file, or using the *rosrun* command. Before starting a Node, a *roscore* must be running. A roscore instance is automatically launched when executing a launch file or can be manually started using the roscore command below.

```
root@ubuntu:~/catkin_ws$ roscore
```

After a roscore has be started, a node can be started using rosrun, an example of which is provided below. This code is run in the top level of the folder above the src file.

```
root@ubuntu:~/catkin_ws$ rosrun qcar_guidance example_guidance_node.py
```

To make the file executable, navigate to the file in the terminal and use the command *'chmod +x filename'*. Further, a list of all running nodes can be provided by running the command *'rosnode list'*. The example above will require a running gazebo simulation however.

3.3 ROS and C++

Using ROS with C++ is referred to as ROSCPP. Files using ROSCPP must use the `#include "ros\ros.h"` call. The ROS library can then be accessed using the `ros` namespace.

Listing 8: CMakeLists.txt ros_example_files package.

```
137 add_executable(imu_test_node src/imu_test.cpp)
138 target_link_libraries(imu_test_node ${catkin_LIBRARIES})
139
140 add_executable(motor_test_node src/motor_test.cpp)
141 target_link_libraries(motor_test_node ${catkin_LIBRARIES})
142
143 add_executable(steering_test_node src/steering_test.cpp)
144 target_link_libraries(steering_test_node ${catkin_LIBRARIES})
```

Unlike python files, C++ files must also be compiled into an executable. This is conveniently handled by the `catkin_make` command, provided the files that you wish to compile are specified in the *CMakeLists.txt* file respective to each ROS package. Listing 8 shows a snippet from the *CMakeLists.txt* file from the *ros_example_files* package. The specified name for the output file, in the example this is the name suffixed with `_node`, can then be used to run the executable with the `rosrun` command or referred to in ROS launch files.

3.4 ROS Topics

ROS Topics are messages used to send data between nodes. Topics all have message types that specify the type and structure of the data being transmitted. Topics are "published" by nodes and can be accessed in other nodes by "subscribing" to the topic.

3.4.1 Publishers

Publishers are used to continuously broadcast a topic over the ROS network. An example of a publisher within Python can be seen in the following code which can be found in *UON-QCAR-BASE/src/ros_example_files/publisher_example.py*.

Listing 9: Python Publisher Example.

```python
#! /usr/bin/env python3
# Every python ROS Node has this first line to ensure script is executed as a python script
import rospy
from std_msgs.msg import Float64

# import rospy which is the python package that deals with ros commands, ...
# eg Subscribers, publishers and ros rates
# import message types so that they can be directly accessed rather than typing std_msgs.Float64

if __name__ == '__main__':
        rospy.init_node('publisher_demo_node')
        #intialises the ros node

        topic_publisher = rospy.Publisher('random_topic/random_subtopic', Float64, queue_size = 0)
        # sets up the publisher, the topic name used can be any name you decide
        # message type can be one of the ros standard messages or you ...
        # can create custom messages using online tutorials
        # queue size dictates amount of messages allowed to queue if subcriber isn't receiving them

        index = 0
        rate = rospy.Rate(1)
        #rate.sleep is used to sleep a node for a brief period of time, the 1 sets ...
        # the frequency of the node
        while not rospy.is_shutdown():
                topic_publisher.publish(index)
                #publish index to the set topic
                index += 1

                rate.sleep()
                #pauses the node so message updates every 1 second as per rate
```

In this code, the variable *index* is incremented every second and published to the topic *random_topic/random_subtc* every second. To ensure the topic is being published, *'rostopic list'* can be used to ensure the topic is in the published topic list.

```
root@ubuntu:~/catkin_ws$ rostopic list
```

```
/random_topic/random_subtopic
/rosout
/rosout_egg
```

Further, the data within this message can be viewed using *'rostopic echo -n1 /random_topic/random_subtopic'* to view the topic data only once.

```
root@ubuntu:~/catkin_ws$ rostopic echo -n1 /random_topic/random_subtopic
```

```
data: 217.0
---
```

3.4.2 Subscribers

Subscribers are used to subscribe to a topic and receive the topic data everytime the topic is published to. Subscribers use a callback function that is called everytime a new message is published to the topic and is used to access the published topic data. An example of a subscriber within Python can be seen in the following code which can be found in *UON-QCAR-BASE/src/ros_example_files/subscriber_example.py*.

Listing 10: Python Subscriber Example.

```
#! /usr/bin/env python3
import rospy
from std_msgs.msg import Float64

class SubscriberClass(object):
   def __init__(self):
      super().__init__() # initalises the class using super instead of class name
      self.subscriber() # intialises the subscriber for the required topic
      self.index_variable = 0.0 # variable that will be grabbed from publisher

   def subscriber(self):
      self.index_sub = rospy.Subscriber('random_topic/random_subtopic', Float64, self.index_callback)
      # Subscriber for set topic, topic name and data type are created in the publisher
      # Subscribers have callbacks that activate everytime the message is published

   def index_callback(self, value):
      self.index_variable = value.data
      print("Published Index:", self.index_variable)
      # callback saves the published data to a class variable

if __name__ == '__main__':
   rospy.init_node('subscriber_demo_node')   #intialises the ros node
   SubscriberClass() #initalises the class
   rospy.spin()   # keeps python from exiting until node is closed
```

In this code, a subscriber class is created that subscribes to the *'random_topic/random_subtopic'* that has a data type of Float64. The subscriber also links to the *index_callback* function that is called when a new message is received. This callback stores and prints the incoming data.
The result of running this script whilst the publisher example is running is shown below.

```
root@ubuntu:~/catkin_ws$ source devel/setup.bash
root@ubuntu:~/catkin_ws$ rosrun ros_example_files subscriber_example.py
```

```
Published index: 1.0
Published index: 2.0
Published index: 3.0
Published index: 4.0
Published index: 5.0
Published index: 6.0
Published index: 7.0
```

3.4.3 Custom Message Types

Within your project, there may come a time were you would like to create a custom message type that is able to publish all the required data in one message. For example, the custom trajectory message used within the example guidance node can be seen below and is provided in *UON-QCAR-BASE/src/qcar_guidance/msg/TrajectoryMessage.msg*.

Listing 11: Custom Message Example.

```
1 float64[] waypoint_times
2 float64[] waypoint_x
3 float64[] waypoint_y
4 float64 velocity
```

This message allows the user to send a list of floats for the waypoint_times, waypoint_x, and waypoint_y variables and a final single float for velocity.
The steps used to create a custom ROS message were created following this tutorial:

- https://medium.com/@lavanyaratnabala/create-custom-message-in-ros-52664c65970d

The following are the steps used to create and use a custom ROS message.

1. navigate to the required ROS package inside your workspace and create a **msg** folder.

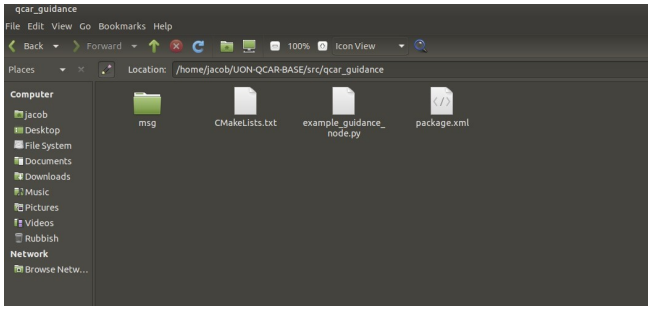


Figure 10: qcar_guidance folder with msg folder created.

2. Within the msg folder, create a file with your message name and the .msg file extension.

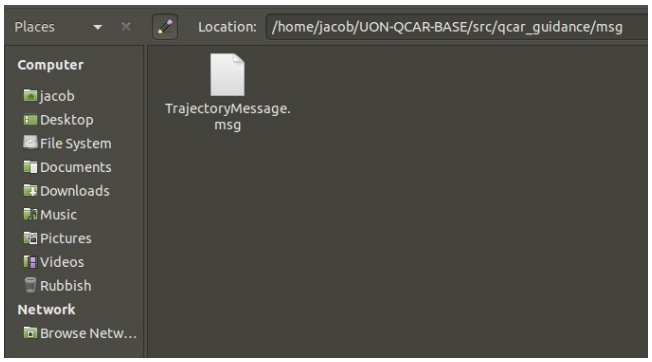


Figure 11: TrajectoryMessage.msg file in folder.

3. Inside this file, create your custom message using the data types used with ROS. These data types are listed here: http://wiki.ros.org/msg

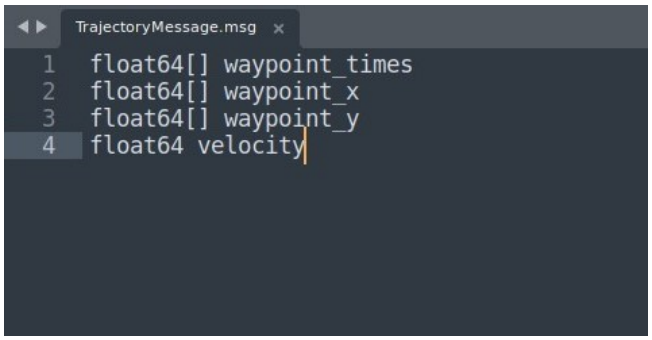


Figure 12: TrajectoryMessage.msg file contents.

4. Open the *package.xml* file and add the lines:
   - `<build_depend>message_generation</build_depend>`
   - `<exec_depend>message_runtime</exec_depend>`

   So that the package.xml contains the following:

```
47    <!-- Use test_depend for packages you need only for testing: -->
48    <!--   <test_depend>gtest</test_depend> -->
49    <!-- Use doc_depend for packages you need only for building documentation: -->
50    <!--   <doc_depend>doxygen</doc_depend> -->
51    <buildtool_depend>catkin</buildtool_depend>
52    <build_depend>message_generation</build_depend>
53    <exec_depend>message_runtime</exec_depend>
54
```

Figure 13: package.xml file with message_generation and message_runtime lines added.

5. Open the CMAKELists.txt file and add message_generation and std_msgs to the catkin REQUIRED COMPONENTS section so that it resembles the follwing image.

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  message_generation
  std_msgs
)
```

Figure 14: catkin REQUIRED COMPONENTS with added contents.

6. Uncomment the add_message_files section and add FILES and your .msg file name so that it resembles the following image.

```
# Generate messages in the 'msg' folder
add_message_files(
  FILES
  TrajectoryMessage.msg
)
```

Figure 15: add_message_files with added msg file.

7. Uncomment the generate_messages section and add DEPENDENCIES and your std_msgs so that it resembles the following image.

```
## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs  # Or other packages containing msgs
)
```

Figure 16: add DEPENDENCIES section with added contents.

8. Within the catkin_package section add CATKIN_DEPENDS message_runtime so that it resembles the following image.

```
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES qcar_guidance
 CATKIN_DEPENDS message_runtime
#  DEPENDS system_lib
)
```

Figure 17: catkin_package with added contents.

9. Re-build the workspace using *catkin_make* to ensure the message has been setup for use correctly.
10. Create a test file to publish the message and view the output. An example of publishing the custom *TrajectoryMessage.msg* is seen below.

Listing 12: Python Publish Custom Message.

```
from qcar_guidance.msg import TrajectoryMessage
... # rest of code here
trajectory_publisher = rospy.Publisher('/qcar/trajectory_topic', TrajectoryMessage, queue_size = 0)
rate = rospy.Rate(0.2)
   while not rospy.is_shutdown():
      trajectoryTopic = TrajectoryMessage() # initialise trajectory message
      trajectoryTopic.waypoint_times = waypoint_times
      trajectoryTopic.waypoint_x = midpoints[0]
      trajectoryTopic.waypoint_y = midpoints[1]
      trajectoryTopic.velocity = velocity
      trajectory_publisher.publish(trajectoryTopic) # publish trajectory message
      rospy.loginfo("Trajectory Published")
      rate.sleep()
```

```
jacob@jacob-virtual-machine:~/UON-QCAR-BASE$ rostopic echo  -n1 qcar/trajectory_
topic
waypoint_times: [0.0, 1.0, 2.0]
waypoint_x: [3.0, 4.0, 5.0]
waypoint_y: [6.0, 7.0, 8.0]
velocity: 0.2
---
```

Figure 18: Custom message output.

This example message and files can be viewed in the *UON-QCAR-BASE/src/qcar_guidance* folder.

3.5 ROS Services

ROS Services are methods of requesting and receiving data from node to node. For most communication, the publisher subscriber method is the easiest and most effecitve communication method. However, some of the pre-made ROS services are useful for grabbing required data in a quick fashion. An example of this service is seen below.

Listing 13: Gazebo Service Python Example.

```
from gazebo_msgs.srv import GetModelState
from gazebo_msgs.srv import GetWorldProperties
import rospy
...   #rest of code goes here
model_names_get = rospy.ServiceProxy('/gazebo/get_world_properties', GetWorldProperties)
#initalise the service before using
model_names = model_names_get()
#call the service to get all model names in gazebo
qcar_state_get = rospy.ServiceProxy('/gazebo/get_model_state', GetModelState)
qcar_state = qcar_state_get("qcar", "")
```

This example shows how to two of the premade Gazebo Services, a list of which are provided here: http://wiki.ros.org/gazebo. The result of this script can be seen in the image below.

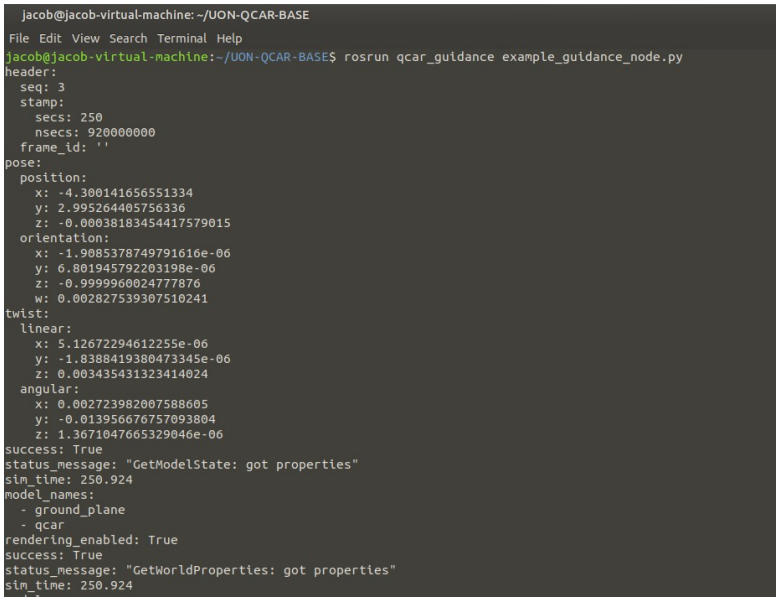


Figure 19: Example rosservice output.

A more in depth example of how to grab the data from this service response can be seen in the file: *UON-QCAR-BASE/src/qcar_guidance/example_guidance_node.py*

4 Gazebo Basics

4.1 World Files

.world files are files that detail the physics and models of a saved Gazebo world. To create world files, open an empty gazebo world and use the insert tab to add models to the world.

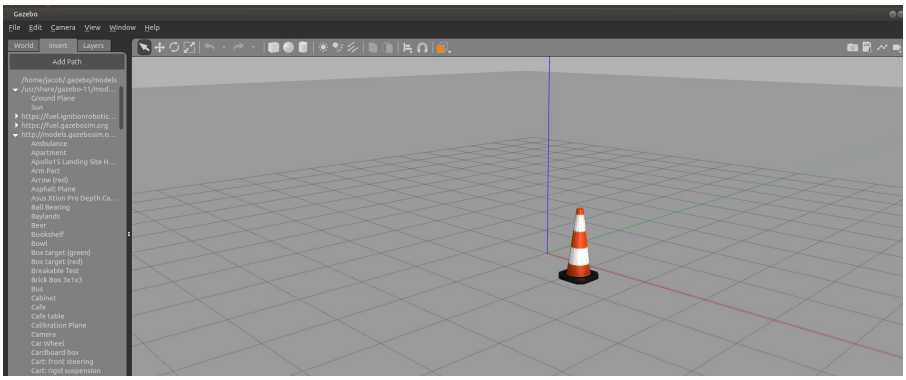


Figure 20: Example Gazebo World.

Once your world is setup, hit file → save world as. Due to an error in the VM and gazebo, this will produce a blank screen that requires gazebo to be minimized and then maximized to refresh. Refresh the screen after every step until saved. This method allows for the manual placing of models within the world.

4.2 QCar Track Builder Script

A better way of designing tracks for the QCar system is to use a script that spawns in required cone models at specified positions. The track spawner script, provided in *UON-QCAR-BASE/src/ros_example_files/track_spawner.py*, takes in cone locations specified within a text file and spawns cones at the inputted coordinates. An example of the required text file format is provided below, with cone colours and x and y positions.

track_example.txt

home > jacob > UON-QCAR-BASE > src > ros_example_files > track_example.txt

You, now | 2 authors (Jacob and others)

```
x y

Blue Cone Positions
 1.00287000e+00  8.99199000e+00
 -4.00900033e-03  8.99858000e+00
 -1.00000000e+00  9.00000000e+00
 -2.00000000e+00  9.00000000e+00
  ....
  2.00000000e+00  9.00000000e+00

Yellow Cone Positions
 1.00044   7.97449
 -0.033975  7.95567
 -1.        8.
 -2.        8.
 ....
2.        8.

Orange Cone Positions
1.1 9.4
1.1 7.6
1.4 9.4
1.4 7.6
```

Figure 21: Track_example.txt format example.

4.2.1 How To Use

1. Edit text file with cone positions required
2. Edit the three cone.sdf files located in *UON-QCAR-BASE/src/qcar_gazebio/models/cones* and change the uri tag value to the correct path to the .dae files within your workspace (i.e change *jacob* to your user name).
3. Within a new terminal, move into and source the workspace and then launch an empty world using the launch file commmand below.

```
root@ubuntu:~/catkin_ws$ roslaunch qcar_gazebo clean_world.launch
```

   This launch file opens an empty gazebo environment as well as a roscore.

4. Within another new terminal, move into and source the workspace and then launch the track spawner node using the command below.

```
root@ubuntu:~/catkin_ws$ rosrun ros_example_files track_spawner.py
```

   This will spawn in the blue, yellow, and orange cones in the x and y positions provided in the text file, an example of which is seen below.

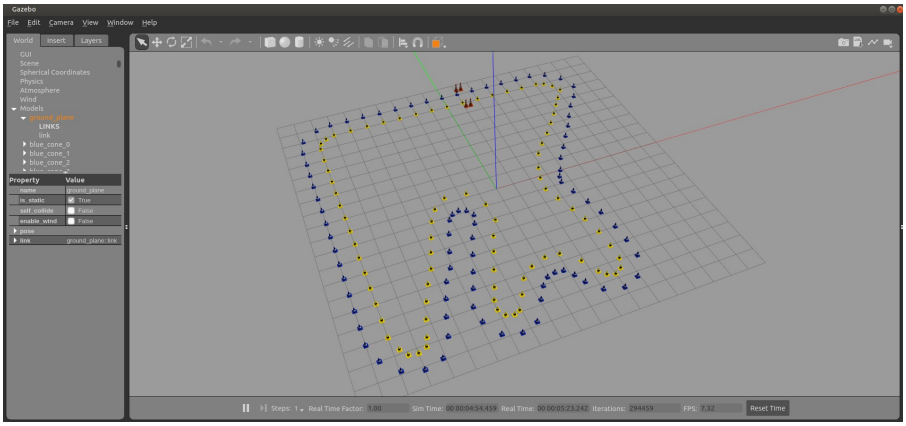


Figure 22: Example Track Designed Using Track Spawner.

5. Save the world for futher use.

5 QCar Model and Simulated Sensors

5.1 QCar URDF Files

The Unified Robotics Description Format (URDF) is an XML format used to describe robots and how their various joints and elements look and act togther. Within this project, the URDF files for the QCar are specified within *.xacro* files. The xacro model files of the QCar used within this project were provided by Quanser and can be downloaded using this link:

- QCar Files: https://quanserinc.box.com/shared/static/uy6088x516p3szf3b07rfknt25yf3eed.zip

Within this URDF system there are three main tags for describing parts of the robot:

- Link
- Joint
- Transmission

Link tags specify a rigid body component of the robot including the visual, collision, and inertial properties of this body. An example of this from the Quanser QCar xacro is provided below.

Listing 14: Base Link Tag.

```
<link name="base">
    <visual>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <geometry>
            <mesh filename="package://qcar_gazebo/models/qcar/QCarBody.stl" scale="0.001 0.001 0.001"/>
        </geometry>
    </visual>
    <collision>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <geometry>
            <mesh filename="package://qcar_gazebo/models/qcar/QCarBody.stl" scale="0.001 0.001 0.001"/>
        </geometry>
    </collision>
    <inertial>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <mass value="0.1"/>
        <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
    </inertial>
</link>
```

Within this link tag, the visual is used to specify the visual geometry, being the QCarBody.stl file provided by Quanser, and the origin of this visual component. The collision tag specifies the physical geometry of the link and the interial tag provides the mass and inertial properties of the component. Joint tags specify the kinematic relationship between the connection two links. An example of this tag is provided below.

Listing 15: Joint Tag Example.

```
<joint name="base_hubfl_joint" type="revolute">
    <parent link="base"/>
    <child link="hubfl"/>
    <origin rpy="0 0 0" xyz="0.12960 0.05590 0.03338"/>
    <axis xyz="0 0 1"/>
    <limit lower="-0.5236" upper="0.5236" effort="300" velocity="2"/>
</joint>
```

The joint tag specifies the type of joint, which are further outlined here: http://wiki.ros.org/urdf/XML/joint, the parent and child link of the joint, the joint origin, the axis on which the joint acts, and the joint limits.
Transmission tags define actuators and how they act on the joint they are attached to. An example is provided below.

Listing 16: Transmission Tag Example.

```
<transmission name="base_hubfr_tran">
    <type>>transmission_interface/SimpleTransmission</type>
    <joint name="base_hubfr_joint">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
</transmission>
```

Within this transmission tag, the type tag specifies the transmission type, the joint specifies the joint the actuator acts on, hardware interface outlines what the actuator will be controlling, in this case the position of the front right hub. To view the full URDF, open the *UON-QCAR-BASE/src/qcar_gazebo/URDF/qcar_model.xacro* file.

5.2 Gazebo Plugins

The Gazebo simulation environment supports the use of plugins to build upon the Gazebo simulator. Plugins enhance *URDF* models, giving them greater functionality. They are also capable of utilising ROS messages and service calls for a variety of applications such as sensors and actuators. These plugins are referenced from within a *URDF* file describing a model and can attach to ROS, modifying models, sensors and visuals each through their respective APIs. All simulated sensors included within the simulation are installed via a variety of plugins, all of which can be turned on/ off by commenting out their respective include lines.

5.3 Inertial Measurement Unit

A simulated IMU sensor is included as part of the QCar simulation to represent the real IMU on the Quanser QCar. This is done through the use of the *IMU sensor (GazeboRosImuSensor)* plugin. The plugin is defined in *UON-QCAR-BASE/src/qcar_gazebo/urdf/imu.xacro*, where the `reference` attribute of the `<gazbeo>` tag defines the IMU's link on the QCar model. The IMU sensor's data is accessed via the ROS topic */imu* defined in line 10 of the plugin. As with any ROS topic, the *imu* topic can be accessed via a bash terminal using the command `rostopic echo /imu`, with the expected output given below.

```
header:
seq: 0
stamp:
secs: 73
nsecs: 290000000
frame_id: imu_link
orientation:
x: 6.7070305049037215e-06
y: 5.79264266238986e-07
z: -0.9999994760267116
w: 0.0010236703485831616
orientation_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
angular_velocity:
x: -2.2946221612156774e-05
y: 0.023369125535142036
z: 0.00021631776525423403
angular_velocity_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
linear_acceleration:
x: -0.00013266481367351358
y: -9.811179371151 01e-06
z: 9.800004333542645
linear_acceleration_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
```

The message is organised into a series of headers which separates the contents of the message. For example, the `angular_velocity` header contains the angular velocity data in x, y and z. This also defines how the data is accessed in code (C++ and Python), being a structure as shown in line 3 of Listing 17, depicting example file *UON-QCAR-BASE/src/qcar_gazebo/ros_example_files/imu_test.cpp*.

Listing 17: IMU message structure.

```
1  void imuCallback(const sensor_msgs::Imu::ConstPtr& msg)
2  {
3      std::cout << "angular velocity in z: " << msg->angular_velocity.z << "\n";
4  }
5
6  int main(int argc, char **argv)
7  {
8      ros::init(argc, argv, "imu_test_node");
9      ros::NodeHandle n;
10    ros::Rate loop_rate(20);
11
12     ros::Subscriber imuSub = n.subscribe("/imu", 1, imuCallback);
13
14     ros::spin();
15
16     return 0;
17 }
```

This is the bare minimum required to subscribe to a node in *roscpp*. As is shown, the ROS node must first be initialised and a node handle defined. A rate in Hz is then set, which defines the update rate of the node. All subscribers (and publishers) are then defined using the node handle as shown. The subscribe method is used to define the subscriber to the */imu* topic, the message queue, 1, and finally the callback function `imuCallback`, which will be called each time a message is received. Finally, the plugin also allows for the inclusion of added Gaussian Noise to simulate measurement noise. This can easily be turned on from within the plugin's definition on line 14 in the `<gaussianNoise>` tags.

5.4 Motors and Encoders

To simulate the DC motor present on the QCar the *Gazebo ROS motor* plugin is installed. This plugin currently includes two plugins, one which simulates an ideal speed controller, and another which models a DC motor, both of which also include a simulated encoder. By default, the ideal speed controller plugin is installed to simplify the simulation. However, this can easily be changed by replacing the joint definitions for each wheel in file *UON-QCAR-BASE/src/qcar_gazebo /urdf/qcar_model.xacro*. Listing 18 below shows the definition of the motor joint for the front left wheel on the QCar. Further information can be found in *UON-QCAR-BASE/src/gazebo_ros_motors-master/README.md*

Listing 18: motor joint for front left wheel.

```
1   <xacro:joint_motor motor_name="wheelfl_motor" parent_link="hubfl" child_link="wheelfl">
2       <xacro:property name="params_yaml" value="$(find gazebo_ros_motors)/params/joint_motor.yaml"/>
3       <origin xyz="0 0 0" rpy="0 0 3.14"/>
4       <axis xyz="0 1 0" rpy="0 0 0"/>
5   </xacro:joint_motor>
```

As the installed plugin is an ideal speed controller, the motors attached to each wheel can easily be set to any angular velocity by simply publishing to the topic */motor/command* replacing *'motor'* with

'wheelrl_motor', etc. Listing 19 shows an example of how to send a velocity to the rear left motor, being example file *UON-QCAR-BASE/src/ros_example_files/motor_test.cpp*.

NOTE: The QCar will not move as only one wheel will be rotating in this example. If you zoom into the car you will be able to see the wheel moving.

Listing 19: Publishing to the motors.

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
#include "std_msgs/Int32.h"
#include "stdio.h"

void callbackEncoder(const std_msgs::Int32::ConstPtr& msg)
{
    std::cout << "Encoder count: " << msg->data << "\n";
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "motor_test_node");
    ros::NodeHandle n;
   ros::Rate loop_rate(20);

    ros::Publisher pubCmdRl = n.advertise<std_msgs::Float32>("/wheelrl_motor/command", 1);
    ros::Subscriber subRl = n.subscribe("/wheelrl_motor/encoder", 1, &callbackEncoder);

    float velocity = 15; // rad/s
    std_msgs::Float32 velCmdL;
    velCmdL.data = velocity;

    while(ros::ok())
    {
        pubCmdRl.publish(velCmdL);

        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

Similar to *subscribers*, *publishers* are defined and set as a method from the node handle `n`. The message type to be published must also be defined, in this case `std_msgs::Float32`. When publishing data a message data type variable must also be defined, which stores the data in the `data` member of the structure. Finally, the message is published using the publish `method` at the rate which is defined. For each motor, the simulated encoder can be accessed by subscribing to the */motor/encoder* topic. This can be seen above in Listing 19 where a `ros::Subscriber` object is created. Finally, a *callback* function `callbackEncoder` is passed by pointer to the subscriber, where the encoder count is printed.

5.5 Steering

The simulated QCar can be steered by utilising the *qcar_controlplugin*. This plugin includes *Joint Position Controllers* which interface with the front wheels to mimic an ideal steering mechanism. This is installed by defining a transmission interface on each wheel hub joint as is shown below in Listing 20. Additionally, an *Effort Joint Interface* is also defined on the wheel joint.

Listing 20: Position Joint Interface.

```
<transmission name="base_hubfl_tran">
    <type>>transmission_interface/SimpleTransmission</type>
    <joint name="base_hubfl_joint">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="base_hubfl_motor">
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<transmission name="hubfl_wheelfl_tran">
    <type>>transmission_interface/SimpleTransmission</type>
    <joint name="wheelfl_motor">
        <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="hubfl_wheelfl_motor">
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
```

The angle of the front wheels is then controlled by publishing to the topic *qcar/base_fl_controller/command* or *qcar/base_fr_controller/command* depending on which wheel is being controlled. This can be seen in Listing 21.

Listing 21: Steering example.

```
{
    ros::init(argc, argv, "steering_test_node");
    ros::NodeHandle n;
    ros::Rate loop_rate(20);
    ros::Publisher pubAngL = n.advertise<std_msgs::Float64>("qcar/base_fl_controller/command", 1);
    std_msgs::Float64 angCmd;

    for(float angle = -30; angle <= 30; angle++)
    {
        angCmd.data = angle*M_PI/180.0;
        pubAngL.publish(angCmd);
        ros::spinOnce(); loop_rate.sleep();
    }
    return 0;
}
```

5.6 Lidar

The simulated Lidar on the QCar is setup using the *"libgazebo_ros_laser.so"* plugin. To enable the lidar in the gazebo simulation, uncommented the include camera.xacro line in the *UON-QCAR-BASE/src/qcar_gazebo/urdf/qcar_model.xacro* file.
The plugin setup is shown below.

Listing 22: Lidar Plugin Setup.

```
<?xml version='1.0'?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <gazebo reference="lidar">
        <sensor name="laser" type="ray">
            <pose> 0 0 0 0 0 0 </pose>
            <visualize>true</visualize>
            <update_rate>10</update_rate>
            <ray>
                <scan>
                    <horizontal>
                        <samples>300</samples>
                        <min_angle>-3.14</min_angle>
                        <max_angle>3.14</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.3</min>
                    <max>18</max>
                </range>
            </ray>
            <plugin name="laser_controller" filename="libgazebo_ros_laser.so">
                <ros>
                    <argument>~/out:=scan</argument>
                </ros>
                <output_type>sensor_msgs/LaserScan</output_type>
                <frameName>lidar</frameName>
                <topicName>scan</topicName>
            </plugin>
        </sensor>
    </gazebo>
</robot>
```

Within this plugin, the gazebo reference refers to the link on the qcar model that the simulated sensor is attached to and the sensor type tag refers to the type of sensor being simulated, in this case a ray sensor. The topicname the lidar data is outputted to is the /scan topic. Parameters such as mininimum and maximum angle and distance, update rate, and visualisation in gazebo can be changed as required. With the camera.xacro included, launching the qcar should produce the following result.

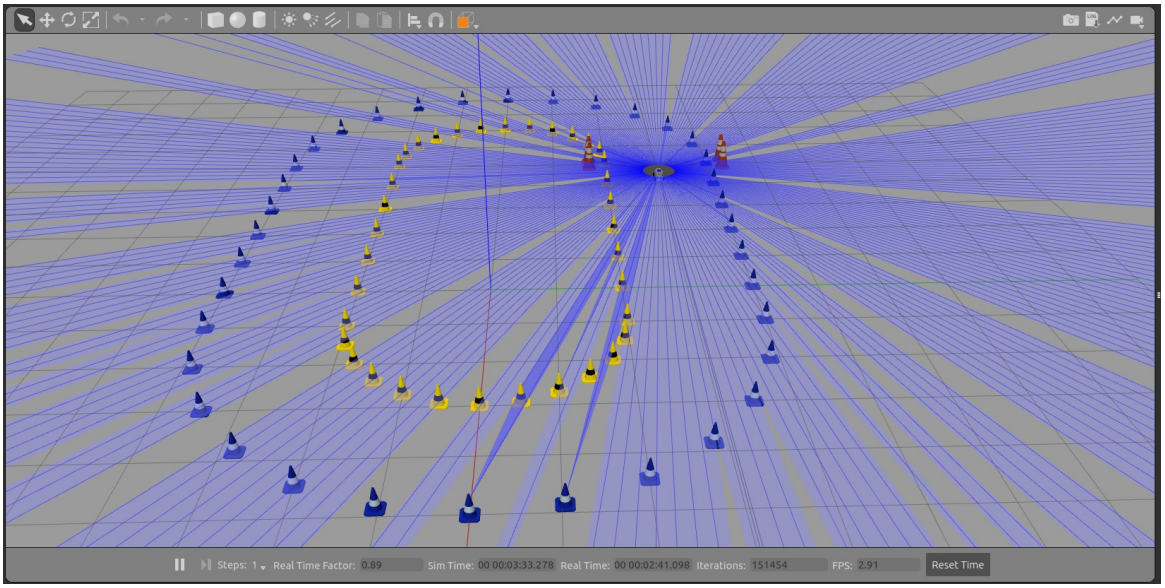


Figure 23: Lidar Visualised In Gazebo Simulation.

The example files on how to retreive and use the outputted lidar data are *lidar_data.py* and *lidar_test.py* in the *UON-QCAR-BASE/src/ros_example_files* folder. The *lidar_data.py* provides a class that subscribes to the lidar message and the *lidar_test.py* provides a way of visualising the retreived data. These files can be seen below.

Listing 23: Lidar Subscriber Class Python.

```python
#! /usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan

class LidarData(object):
  def __init__(self):
    super().__init__()
    self.subscriber()
    self.angle_min = 0.0
    self.angle_max = 0.0
    self.range_min = 0.0
    self.range_max = 0.0
    self.ranges = []
    self.angle_increment = 0.0

  def subscriber(self):
    self.lidar_sub = rospy.Subscriber('/scan', LaserScan, self.callback, queue_size=1, buff_size=2**24)

  def callback(self, value):
    self.ranges = value.ranges
    self.angle_min = value.angle_min
    self.angle_max = value.angle_min
    self.range_min = value.range_min
    self.range_max = value.range_max
    self.angle_increment = value.angle_increment

  def get_angle_data(self):
    return self.angle_min, self.angle_max, self.angle_increment

  def get_ranges(self):
    return self.ranges
```

This subscriber grabs the relevant information from the lidar topic including the minimum and maximum for the angles and ranges, the detected distance and the angle increment between the rays. The *lidar_test.py* file uses this class and visualises the incoming lidar data, as shown below.

Listing 24: Lidar Test File Python.

```python
#! /usr/bin/env python3
import rospy
import lidar_data
from time import sleep
import math
import matplotlib.pyplot as plt

if __name__ == '__main__':
    rospy.init_node('lidar_node', disable_signals=True)
    rate = rospy.Rate(0.2)
    scan_sub = lidar_data.LidarData() #create lidar data class from lidar_data.py file
    rate.sleep()
    angle_min, angle_max, angle_increment = scan_sub.get_angle_data()
    ranges = scan_sub.get_ranges()
    plt.figure()
    current_angle = angle_min
    plot_angle = 0.0
    plt.plot(0,0,marker="o",markersize=5,markerfacecolor="black", markeredgecolor="black")
    for i in range(0,len(ranges)):
        if ranges[i] != math.inf:
            x = ranges[i]*math.cos(current_angle+math.pi/2)
            y = ranges[i]*math.sin(current_angle+math.pi/2)
            plt.plot(x,y,marker="o", markersize=5, markerfacecolor="green", markeredgecolor="green")
        current_angle = current_angle + angle_increment
    plt.show()
```

This file runs from the minimum to maximum angle limit and plots any detected points within a graph for display. The addition of *math.pi/2* is added for plotting of points within python. When the *lidar_test.py* file is run, the output should resemble the following that shows the obstacle locations surrounding the qcar.

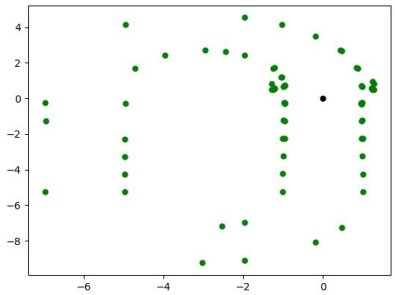


Figure 24: Output Of Lidar Test File.

5.7 Cameras

The files *depth_camera.xacro* and *camera.xacro* within the *UON-QCAR-BASE/src/qcar_gazebo/urdf* folder outline the plugins used to simulate the cameras on the QCar. To the cameras within gazebo, uncommented the include *depth_camera.xacro* and the include *camera.xacro* in the *UON-QCAR-BASE/src/qcar_gazebo/urdf/qcar_model.xacro* file.
The file *UON-QCAR-BASE/src/ros_example_files/camera_test.py* provides a method for extracting and viewing the outputs of these cameras and can be seen below.

Listing 25: Camera Test File Python.

```python
#! /usr/bin/env python3
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv_bridge
import cv2
def depth_camera_callback(data):
     try:
          bridge = cv_bridge.CvBridge()
          depth_image = bridge.imgmsg_to_cv2(data, desired_encoding="passthrough")
          cv2.imshow("Depth Image", depth_image)
          cv2.waitKey(1)
     except Exception as e:
          rospy.logerr("Error Message: %s" , str(e))
def camera_callback(data):
     try:
          bridge = CvBridge()
          cv_image = bridge.imgmsg_to_cv2(data, "bgr8")
          cv2.imshow("Camera Image", cv_image)
          cv2.waitKey(1)
     except Exception as e:
          rospy.logerr("Error Message: %s" , str(e))
def main():
     rospy.init_node("camera_viewer", anonymous=True)
     depth_camera_topic = "/depth_camera/depth/image_raw"
     rospy.Subscriber(depth_camera_topic, Image, depth_camera_callback)
     camera_topic = "/front_camera/image_raw"
     rospy.Subscriber(camera_topic, Image,camera_callback)
     rospy.spin()
if __name__ == "__main__":
     main()
```

The *bridge.imgmsg_to_cv2* converts the image message from ros to an OpenCV format that is easily processed within python. Running this file with the camera and depth camera running should produce the following result.

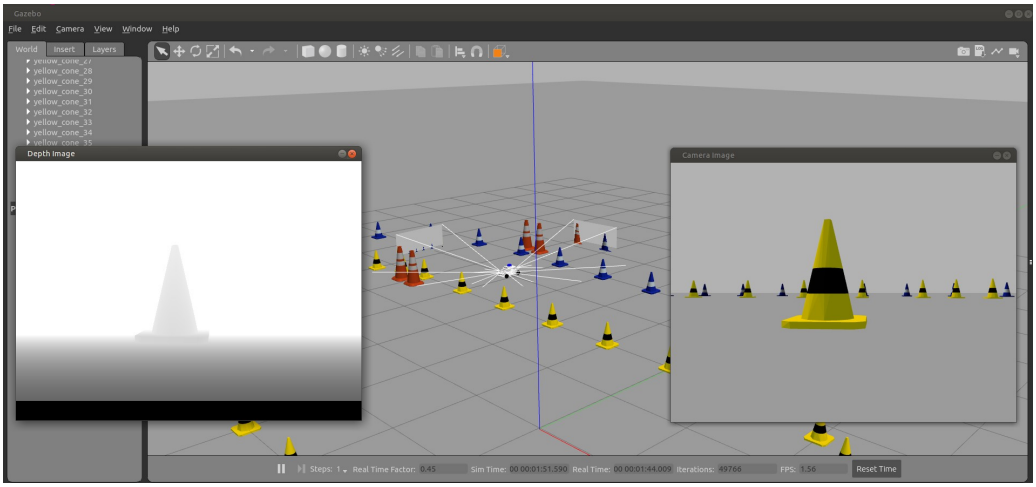


Figure 25: Gazebo Running With Camera Test File.

5.8 Manual Controller GUI

The file *UON-QCAR-BASE/src/ros_example_files/controller_GUI.py* provides a manual method of control of the qcar for testing. To run the file, launch a world with the QCar spawned in and then rosrun the *controller_GUI.py* file. The output should resemble the following image and provide an interface to manually change the speed and steering of the QCar.

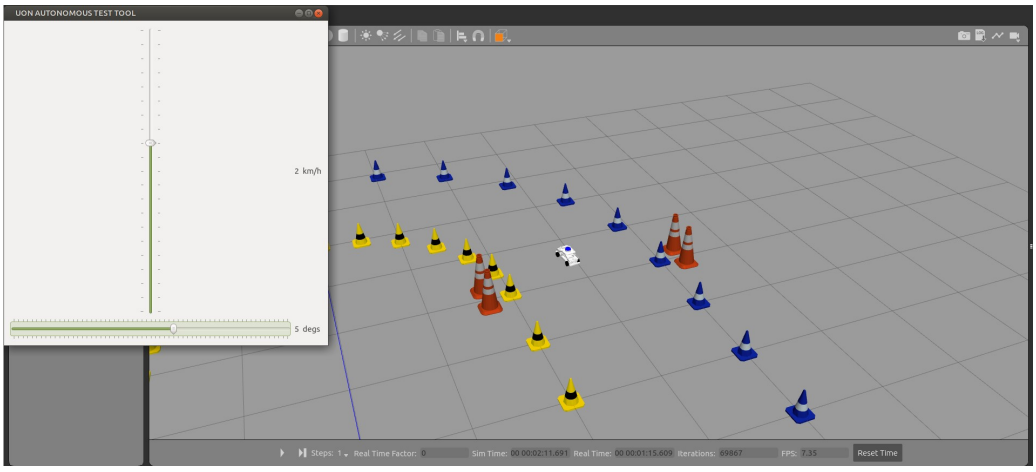


Figure 26: Controller GUI In Use In Gazebo Simulation.

6 Example Files and How to Use Them

As part of this simulation a series of basic examples have been included. This section will briefly outline these examples and how they should be run.

NOTE: Make sure to source the UON-QCAR-BASE workspace prior to running these commands from within the root workspace directory.

6.1 Example Launch Files

A few basic example launch files have been created which allows the QCar to drive around the example tracks. These examples include a basic solution for guidance and control. The examples can be launched with the following command, replacing `Track1` with `Track2` or `Track3` for the desired track.

```
root@ubuntu:~$ roslaunch mybot_gazebo Track1Example.launch
```

The command will launch an instance of Gazebo, spawning into the desired track world. The QCar will be placed at the starting position for each track and will begin to slowly drive around the track. The control node will produce a continuous output detailing the current time, wheel angular velocity and steering angle as is given by the output below.

```
[Control_Node]
[time]: 10.45 secs
[omga]: 9.09 rad/s
[dlta]: 0.44 deg
```

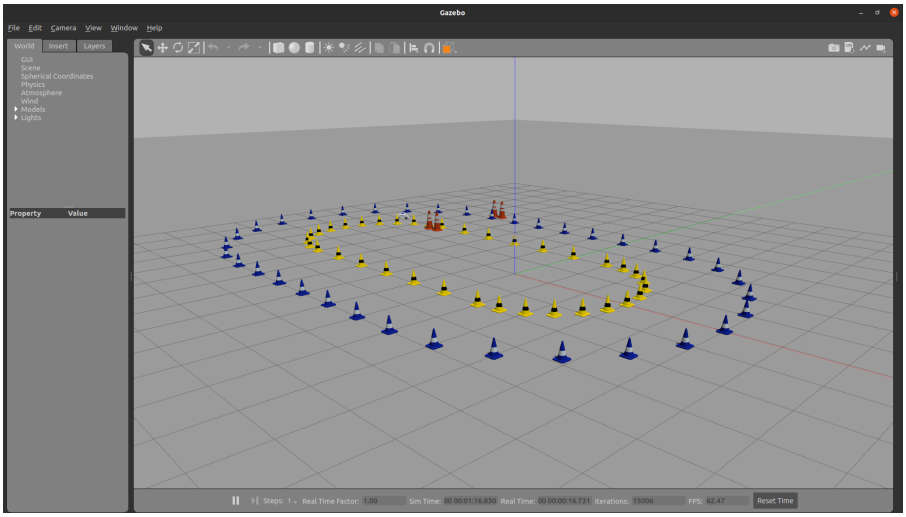


Figure 27: Track1 example simulation.

6.2 Example Nodes

All C++ and Python files included in the *ros_example_files* package can be launched as standalone nodes, provided one of the Track launch files have already been launched. Otherwise they will not operate as intended.

All example nodes can be launched with the following command, where `ros_example_node` refers to the package *ros_example_node*. And `imu_test_node` the node you wish to run. This is a quick way to test your nodes before running them inside launch files with other nodes.

```
root@ubuntu:~$ rosrun ros_example_files imu_test_node
```

ROSCPP example nodes:

- imu_test_node (imu_test.cpp)
- motor_test_node (motor_test.cpp)
- steering_test_node (steering_test.cpp)

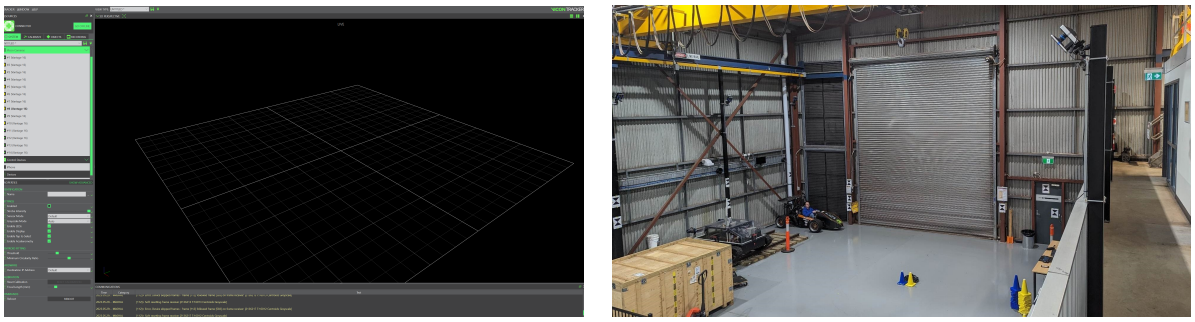
ROSPY example nodes:

- camera_test.py
- get_qcar_state_example.py
- controller_GUI.py
- lidar_data.py
- lidar_test.py
- publisher_example.py
- subscriber_example.py
- track_spawner.py

7 Vicon Camera System

The Vicon camera system can be used as a local GPS system and is a powerful tool for testing real world implementations of autonomous systems. There are two Vicon Camera systems at the University of Newcastle, both of which are located in the Engineering Precinct in the TA building, and ES313 Autonomous Research Laboratory. This setup will refer to the TA building system.

The setup consists of 14 Vicon Vantage V16 cameras, capturing the area given in Figure 28. The cameras track in real-time the position of small retro-reflective markers which can be strategically placed on stationary or moving objects to enable object tracking. Multiple markers can also be placed on one object where they can be manually grouped using the Tracker 3 software, shown below, to form object skeletons. By doing so, the markers can be used to track an object's position and orientation, therefore providing an object's pose. The cameras are connected via two PoE (Power over Ethernet) switches, which are controlled via a Vicon host PC. Additionally, the host PC is also connected to a separate modem allowing for the wireless transmission of all Vicon tracking data.



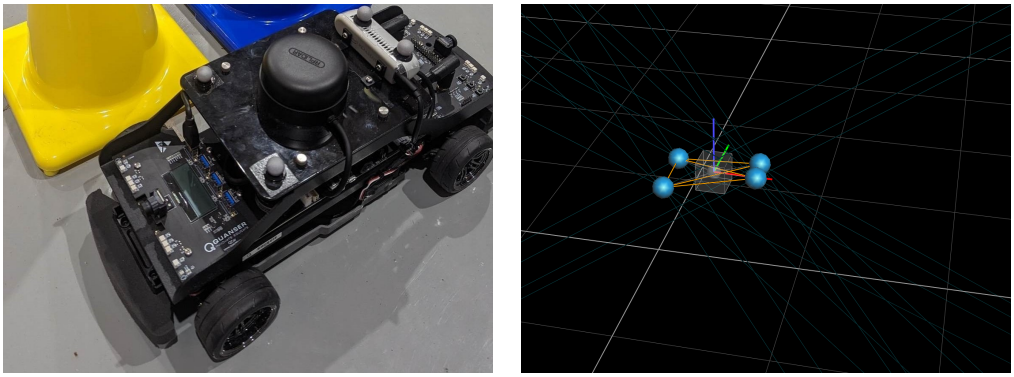
(a) Vicon Tracker 3 software.

(b) Area captured by cameras.

Figure 28: Vicon software and setup.

To track the pose of the QCar specifically, reflective markers can be placed on the top portion of the car's chassis shown below in Figure 29. By doing so, the QCar can be tracked by the Vicon system by grouping the markers to form an *object* within the Tracker 3 software. The data available from the system includes object position and orientation viewable from within the software. In addition, Vicon data is also transmitted via any network connected to the host PC. Therefore, by connecting the QCar to the network created by the modem connected to the host pc, the QCar can gain access to the Vicon data.

To then gain access to this data within ROS, an open-source Vicon Bridge package can be used which advertises a ROS topic with the Vicon tracking data. This topic can be subscribed to, to retrieve said data.

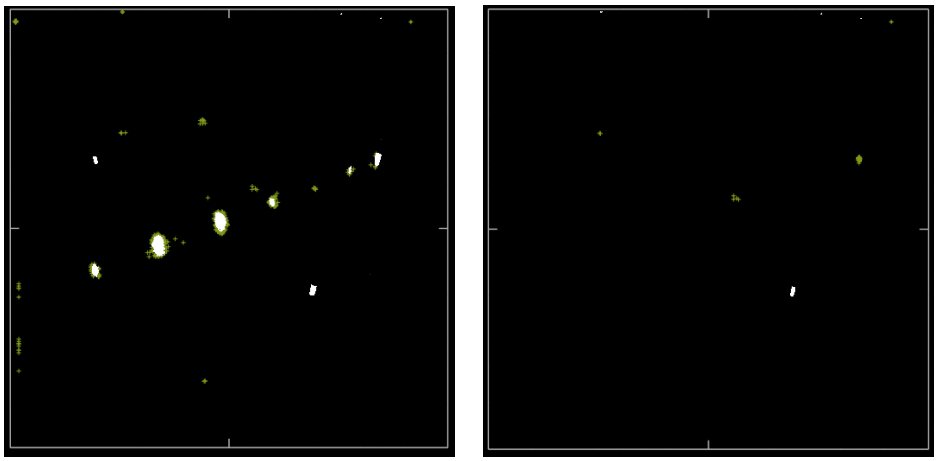


(a) Markers on QCar. (b) QCar virtual Vicon object.

Figure 29: Tracking QCar with Vicon.

7.1 Camera Calibration

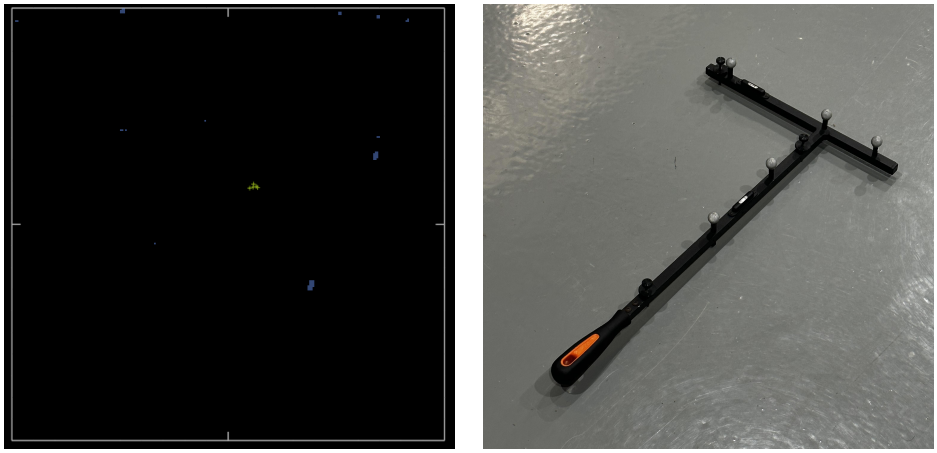
To ensure the tracking data obtained from the Vicon camera system is accurate, the calibration and setup of the camera system is vital. This process can be undertaken with reference to the Vicon documentation, provided next to the Vicon PC. The Vicon calibration process ensures that the system only tracks the markers and does not confuse other sources of reflection with them. Additionally it ensures the visibility of each camera attached to the system.



(a) Before tuning. (b) After tuning.

Figure 30: Vicon Camera Strobe Intensity and Threshold tuning.

First, each camera window should be visually inspected from within the software interface for interference from foreign sources of reflection. Properties including *Strobe Intensity* and *Threshold* (found in the advanced settings) should be tuned for each camera so that only sources of reflection which are known to be markers remain. Figure 30 illustrates the before and after of tuning these values.

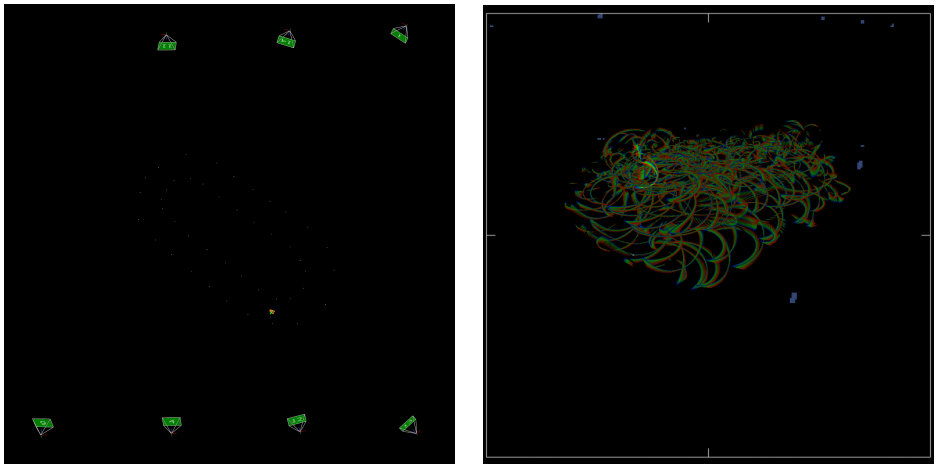


(a) Vicon camera masking. (b) Vicon Vantage calibration wand.

Figure 31: Vicon camera masking unwanted reflections.

To completely eliminate persistent non-marker reflections, the camera masking tool should then be used. This process allows for certain portions of a camera's view to be masked over and ignored when capturing data for image processing. This is shown in Figure 31, where the only markers remaining are those present on the Vicon calibration wand.

Finally, to calibrate the camera system the calibration wand is used which is waved throughout the target volume. During this process each camera tracks the wand where a target number of capture frames must be reached. Additionally, tracks of the wand's path appear within the Tracker 3 software's camera views which should be evenly distributed throughout the volume. Upon successfully completing this calibration, the volume origin and the orientation of axis can be defined using the calibration wand. This process can be seen below in Figure 32.



(a) Final target volume. (b) Camera calibration tracks.

Figure 32: Vicon Camera system calibration.

8 Useful Resources and Links

- Articulated Robotics for Gazebo and ROS tutorials:
  https://www.youtube.com/@ArticulatedRobotics
- Basics on how to setup ROS workspace plus make and control of simple robot: https://www.generationrobots.com/blog/en/robotic-simulation-scenarios-with-gazebo-and-ros/
- Create Custom ros message:
  https://medium.com/@lavanyaratnabala/create-custom-message-in-ros-52664c65970d
- ROS Nodes
  http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes#:~:text=Nodes%3A%20A%20node%20is%20an,helps%20nodes%20find%20each%20other)
- Dual boot Ubuntu to external SSD
  https://medium.com/geekculture/installing-linux-ubuntu-20-04-on-an-external-portable-ssd-and-pitfalls-to-be-aware-of-388294e701b5
- ROS extension for Visual Studio Code
  Extension ID: ms-iot.vscode-ros
- Vicon_bridge. To get vicon data onto the car from the vicon network: https://github.com/ethz-asl/vicon_bridge

Useful tip, to prevent nodes from running whilst ROS/Gazebo is starting up, the following code can be used in Python.

Listing 26: Make Node Wait For ROS Before Starting Python.

```
# prevents the node from running whilst ros/gazebo is starting up
rosrunning = 0
while rosrunning==0:
        sleep(1)
        if rospy.Time.now() != rospy.Time():
                rosrunning = 1
```

8.1 Useful Commands

- `rostopic list` (prints list of current topics)
- `rostopic echo -n1 /topic_name` (echoes topic only once)
- `history` (prints terminal command history)
- `CTRL + R` (search terminal history)
- `roscore` (starts an instance of roscore)