Journal of UAB ECTC
Volume 18, 2019
Department of Mechanical Engineering
The University of Alabama, Birmingham
Birmingham, Alabama USA

AUTONOMOUS VEHICLE SIMULATION USING OPEN SOURCE SOFTWARE CARLA

Emily Barbour, Kevin McFall
Kennesaw State University
Marietta, Georgia USA

ABSTRACT

The topic of autonomous vehicles has grown tremendously in the past 10 years. Research into different methods of computer vision, path planning algorithms, and controls theories have been an area of great interest for the automotive industry. While many of these systems can be theorized off data collected in a driver-controlled environment, the testing of their holistic application remains a challenge for researchers to properly complete in a realistic and safe environment. Thus, computer simulations have been developed to help imitate real environments in such a way that rapid prototyping, training, and validating can be done in a safe, cost effective, and time saving manner,

In this paper, one like simulation, named CARLA, is explored and investigated for its potential to test implementations of algorithms and controls theories in replicable, controlled fashion. Furthermore, the communication framework ROS will be utilized, and the official ros-bridge investigated. Such a system will allow an entire control stack to be simulated, the inner working of which will have no way to distinguish between simulation and real environments, allowing for most of the design to be re-utilized in a real-world model.

KEY WORDS: *CARLA, ROS, Autonomous Vehicles, Simulation, Prototyping*

INTRODUCTION

CARLA is an open-source vehicle simulator targeted at aiding research and development of autonomous vehicle control solutions. As demonstrated in the paper introducing CARLA [1], autonomous vehicles built from many different machine learning algorithms can be tested, allowing for rapid implementation and experimentation of different algorithms in different environments. Furthermore, hazards can also be introduced into the simulated environment, such as differences in lighting conditions, rain, pedestrians, and other vehicles.

CARLA has seen use in the academic field in developing autonomous vehicles, specifically reinforcement learning and conditional imitation learning [2], as their iterative nature lends well to accelerated computer simulation. Furthermore, these algorithms can often be dangerous to train in real life environments, as their first attempts are often unaware of any of the concepts to fulfil their tasks.

One of the key benefits of CARLA is its real-time generation of "ground truth" data. Due to each object in the environment being categorized and tracked, information such as ground truth image-segmentation from a mounted camera becomes a trivial task.

Another key benefit is the configurability of the environment. Any component in the environment can be easily scripted to create specific test-case scenarios to understand how a control solution might behave. Furthermore, these can be packaged together and rerun with no variability to allow for consistent validation and training data to be utilized.

HARDWARE REQUIREMENTS

Before detailing the exact process recommended to install CARLA and ROS, a few notes should be made before beginning the installation process.

First, it should be noted that, many autonomous vehicle solutions require good hardware in order to run at a reasonably fast rate to be usable. With CARLA, however, this requirement can be lowered due to its Synchronous Mode and fixed time step. These allow for the simulation to complete an entire calculation cycle and publish that information, waiting until a tick is received from a client until the next cycle is completed. With the implementation of ROS, this tick can be halted until user-developed algorithms finish their calculations as well, allowing for much slower hardware to be utilized in the simulation.

This does not come without caveats, however. Since the simulation is halted until all calculations are complete, considerably under-powered hardware can vastly increase the time it takes for each user testing iteration. Thus, it is recommended that users have hardware that is able to run the simulation itself in real time, noted in the documentation [3] as at least 10 FPS.

Furthermore, due to the nature of the type of computational load the simulation takes, both the CPU and GPU of the user's machine need to be considered in order to run the simulation in real time.

It is also recommended that a clean installation of the operating system be used, one with at least 50 GB of storage space, to sandbox the development and prevent unintended system package changes from interfering with ROS and CARLA. It is recommended that this be its own drive, but extra steps can be followed to install it alongside another OS.

Since this paper will be working with ROS, a Linux based OS is by far the most popular solution. Ubuntu is the distribution of choice for many and is one of the easiest for users unfamiliar with developing on Linux. As to which version, 16.04 LTS was chosen for a variety of reasons for this paper, mainly due to it being the version which most development packages target for release, and for the number of packages available for its ROS release, ROS Kinetic.

INSTALLATION OVERVIEW

The overview of installing this development environment is as follows:

1. Install Ubuntu 16.04 LTS
2. Install ROS
3. Clone CARLA
4. Link CARLA development packages
5. Install carla-ros-bridge
6. Install any extra dependencies

This paper will take a holistic approach in demonstrating the steps to install and configure the environment.

INSTALLING UBUNTU

Installing Ubuntu requires first writing an installation image onto a bootable media other than the one intended to host the development environment. In most cases, an 8 GB USB flash drive will suffice.

The installation image can be found on the official ubuntu site [4]. A desktop image is recommended due to it installing many graphical packages required for running CARLA. Click the link pointing to either the 64-bit (most common) or 32-bit image, referring to whether the PC intended on running CARLA is 32-bit or 64-bit.

This will download a .ISO file, a common filetype used for storing copies of systems and CD / DVDs. Turning this into a bootable medium requires the use of an additional program. The recommended software for creating a bootable USB is Etcher [5] for Windows, Mac, and Linux. A guide for creating a bootable USB flash drive on Mac has been created by Ubuntu [6]. In short, running Etcher after inserting the USB flash drive into the machine will allow selecting the flash drive and the Ubuntu ISO, in order to create a bootable medium.

Once finished, restart the machine. During the initial splash screen display during boot, before any OS is loaded, press the key on the function row (F1-F12) corresponding to “select boot device.” If the USB flash drive was configured correctly, an entry labeled something like, “USB FLASH DRIVE”, or “BOOTABLE USB”, or “UBUNTU” will display amongst the list of bootable mediums. Select it start up the installation process. Ubuntu will ask the user to either install or to try the OS; selecting install will load the OS with the installation menu displayed.

Standard OS installation follows, including system language, time-zone, computer name, username and password, encryption, etc. Before committing to the install, the installer will prompt the user to pick the installation location. Once selected, any data stored at the location specified is not guaranteed to be recoverable. It is recommended that an entire drive be picked as the installation target, allowing for ubuntu to create the proper partition sizes automatically, but advanced users can specify specific partitions if they wish. Once complete, shut down the computer, remove the USB Flash drive, and boot up the computer, which should now have defaulted to Ubuntu as its default OS.

It is recommended that the machine be allowed time to update all the packages that have pre-installed to their latest versions before continuing.

INSTALLING ROS

ROS has expansive documentation for installation, as well as tutorials on usage and package explanations [7]. As such, specific commands for the installation process can be found on their documentation page, and this paper will cover the overall steps required to install ROS.

The release of ROS being used for this paper is Kinetic, targeted at Ubuntu version 16.04 LTS. The flow of steps is as follows:

1. Add ROS to the list of verified package repositories
2. Install ros-kinetic-desktop-full
3. Initialize rosdep
4. Edit the .bashrc file for ROS commands to be enabled by default
5. Install dependencies for building ROS packages

These steps will install all the packages used for developing with ROS, including their dependencies, and as such will make up around 5 GB of space. Creation of the catkin workspace will be delayed until after installing CARLA and its ros-bridge due to their implementation.

CLONE CARLA

CARLA has provided useful documentation detailing information regarding the basics of running the simulation as well as installation requirements [8]. These will be referenced during the install and setup process. The latest release of CARLA can be found on its github repository [9]. Git is an open-source software version-control system which tracks every change made during software development. Github is the most popular hosting site for git repositories, and most open-source projects utilize the site for its renowned collaboration and backup services.

Using that repository, clone its contents into the Documents folder using the following commands from the Ubuntu command line interface:

```
cd ~/Documents

git clone https://github.com/carla-simulator/carla.git carla
```

Before running any of the examples provided, installation of pygame and numpy is necessary to run the python scripts. Installation can be done using:

```
python -m install --user pygame numpy
```

CARLA should now be successfully installed, and any of the examples in their documentation can be run to demonstrate some of the simulator's capabilities.

LINK CARLA DEVELOPMENT PACKAGES

One final step in ensuring CARLA is ready for development is linking their Python development packages to the system variable PYTHONPATH. By doing so, it allows programs utilizing the CARLA library to be run on the development machine, both as individual scripts and as nodes run through ROS.

This PYTHONPATH variable can be modified in many ways, including manually appending to the variable during the launch of a ROS package. However, the easier and recommended method is to append a command to the end of the .bashrc file located in the home directory. This file can be thought of as a list of commands to be run whenever a bash terminal of any sort is created on the development machine, including any made by ROS nodes and packages.

Before linking, the .egg (a file compression format like tar and zip) containing the CARLA libraries must be located. As of version 0.9.5, the version being used in this paper, the archive is located at:

```
$(CARLA)/PythonAPI/carla/dist/carla-0.9.5-pyX.Y-linux-x8
6_64.egg
```

X.Y is the python version the library was developed for. There are two target versions in the 0.9.5 version of CARLA: Python 2.7 and 3.5. ROS development and communication packages are developed for Python 2.7, therefore this paper utilized the archive:

```
carla-0.9.5-py2.7-linux-x86_64.egg
```

Furthermore, the carla-ros-bridge operates under the assumption that the archive is located at:

```
$(CARLA)/PythonAPI/carla
```

So, a symbolic link also needs to be created so that the archive points to that destination. All of these can be distilled down to one command, which will take the command and append it to the end of the .bashrc file:

```
echo “export PYTHONPATH=$PYTHONPATH:/home/emi
lybarbour/Documents/carla/PythonAPI/carla/dist/carla-0.
9.5-py2.7-linux-x86_64.egg:/home/emilybarbour/Docume
nts/carla/PythonAPI/carla/” >> ~/.bashrc
```

Finally, either sourcing the .bashrc file, or rebooting the terminal will execute that command and allow the CARLA libraries to be successfully imported.

INSTALL CARLA-ROS-BRIDGE

The CARLA ros-bridge [10] can be found from the same author on GitHub as the main CARLA repository. The README contains helpful information about installation, as well as all the messages and integrations available for ROS nodes to interact with.

The exact commands to setup the ros-bridge can be found in the README, but an overview of the steps are as follows:

1. Create the folder structure
2. Clone the repository
3. Link ros-bridge packages into the catkin_ws/src folder
4. Install all required ROS package dependencies found in each ros-bridge package
5. Compile the packages and link them to the ROS workspace

Launching any of the .launch files in the carla_ros_bridge package will successfully connect to the simulator if an instance of the CARLA simulator is already running on the machine.

INSTALL EXTRA DEPENDENCIES

Several extra libraries that can be useful for developing different types of autonomous vehicles are also included:

1. CUDA 9.0 and cuDNN 7.0 [11]: Libraries for developing applications that utilize CUDA cores found on an NVIDIA GPU. Useful for any application that can utilize vector mathematics to vastly speed up calculations, such as LIDAR processing and Machine Learning.
2. ros_numpy [12]: Helpful package to convert ROS Sensor data types to numpy arrays, a fast and efficient way of representing data in python. Placing the repository inside the catkin_ws/src will add its library to ROS packages upon a one-time execution of catkin_make.
3. Point Cloud Library (PCL) [13r]: An opensource library containing many algorithms for filtering and interpreting Point Clouds, the data organization ROS Lidars utilize. Installation can be done by adding the following commands to the CMakeLists.txt file in a dummy ros package inside the catkin_ws:

```
find_package(PCL REQUIRED)
```

```
...
include_directories(${PCL_INCLUDE_DIRS})
...
target_link_libraries(<YOUR_TARGET> ${PCL_LIBRARI
ES})
```

Then ROS will install the package with the following command:

```
rosdep install –from-paths src –ignore-src –rosdistro kinet
ic -y
```

4. additional python packages through pip: matplotlib, scipy
5. additional python packages through apt-get: python-opencv, python-opencv-contrib

INTRODUCTION TO CARLA

With the development environment configured, a coarse overview of CARLA is presented. One key idea to note is that each process in CARLA runs as its own instance, their only communication amongst one another being the Python CARLA library. Because of this, the CARLA simulator, once launched, will need to persist in its own bash terminal, and subsequent terminals will need to be opened to each host their own programs.

In order to launch CARLA, the CarlaUE4.sh bash script needs to be executed. An example command might be:

```
./CarlaUE4.sh -windowed -ResX=320 -ResY=240 -bench
mark -fps=10
```

Any Unreal Engine 4 command can be passed to the bash script. Extra, CARLA specific commands have been added as well. A fixed framerate can be achieved with the fps=X command, and is recommended to set to define the time step between each iteration. For example, launching CARLA with a fixed framerate of 20 on low quality will look like this:

```
./CarlaUE4.sh -fps=10 -quality-level=Low
```

Other important commands feature setting maps, loading scenarios, setting graphics fidelity, and setting the local port over which CARLA-program communications will occur.

Once launched, CARLA will open the map specified, or the default, as in Figure 1. Once the world has been loaded, the simulation is ready for Python programs to communicate.

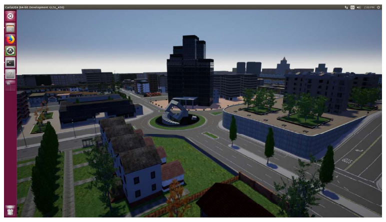

Figure 1. Default Map with no vehicles loaded when running CarlaUE4.sh

Many of the different functions available to developers are demonstrated by the python examples bundled with the CARLA repository. Some of the notable include:

- setting weather conditions, which affect traction and camera sensor data
- setting and changing maps, scenarios
- accessing vehicles in the simulation and reading or modifying properties, such as position, speed
- spawning vehicles, removing vehicles
- controlling traffic lights
- Comparing world positions, generating paths between positions following road laws
- converting between graphical coordinates and geoSat coordinates

ROS INTEGRATION

One of the benefits of ROS is its ability to separate development amongst many packages and create applications that combine those packages together for a specific controls stack. The ROS integration furthers this development ideology by allowing an additional package to be developed alongside a control stack dedicated for a physical vehicle, delivering sensor data to said package in the same way the physical vehicle would. In essence, it allows a system to be “tricked” into thinking CARLA data is real world data, allowing it to be tested and trained without the physical vehicle, and all the danger and limitations that imposes.

In order to have ROS integrate with CARLA, the carla_ros_bridge package needs to be launched alongside the vehicle controls stack and the user developed CARLA integration scripts, all of which can be done within a single launch file. The ros-bridge communicates with CARLA, relaying information such as map, synchronization, simulation time, and basic information about any dynamic object placed in the map. This is illustrated in Figure 2.

Furthermore, any vehicle matching one of the names inside a list will be given a large host of topics ranging from sensor data to vehicle position, as well as the ability to control the vehicle either through a throttle, steering model, or an Ackermann drive model. These topics return simulated sensor data in the standard ROS format, speeding up development times.

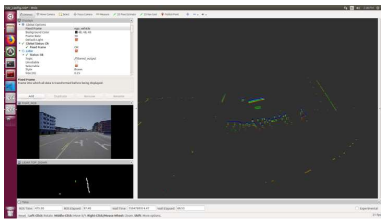

Figure 2. RVIZ Display showing topic information from sensors and diagnostics generated from the ros_bridge. Only ROS controlled vehicle exists in this example.

Another package inside the ros-bridge exists for quickly configuring vehicles and sensor arrays, as well as providing a launch file to quickly spawn the vehicle into the environment for control. Through a json file, a list of sensors, their type, name, position relative to the ground center of the car (in meters). Data created by these sensors can be found in the topics:

```
carla/(VEHICLE_NAME)/(SENSOR_NAME)
```

Finally, a ROS package exists that can convert a pose in the environment to a path from the vehicle to the pose following road laws. This is particularly useful for testing vehicle controls and path following without having to calculate the path itself with sensor data.

EXAMPLE PACKAGE

This next section gives an overview on the specifics to create a ROS package that will:

- Connect the ros-bridge to the CARLA process
- Spawn in an ego-vehicle with a specific sensor array and vehicle type
- Control the vehicle in either a 'Throttle Steer' model or an Ackermann model
- Enable and disable built in autopilot
- Send basic motor commands
- Receive and process LIDAR data coming from the ego-vehicle

LAUNCHING ROS-BRIDGE

First, after creating the catkin package, a launch file will be created to spawn instances of all of the vehicles nodes, as well as ros-bridge. Common organizational practices for ROS dictate a config, include, launch, and src folder be created inside the package to help organize all of the files needed to run the vehicle. Thus, the launch file will be located at:

```
${PACKAGE}/launch/${NAME}.launch
```

While this specific example utilizes a single launch file, separating individual parts into their own launch files helps modularize the system, allowing for multiple launch files to be created to initialize different parts of the system for different environments or testing purposes.

The launch file is written in a markup language akin to XML. As a result, any piece described hereafter can theoretically be placed inside any launch file, as long as the hierarchy is properly respected.

Finally, it is recommended to mask any potentially variable information at the top of the launch file with default values, such that any of them can be quickly modified from the command line without need to modify the launch file. In order to create an argument with a default value, use the following command as the first child of the launch node:

```
<arg name='NAME' default='VALUE'/>
```

where NAME is the name to be referenced both in the command line and in the launch file, and VALUE is the default value to be used if no argument is passed via the command line. Both NAME and VALUE are encased in single quotes. The argument can thereafter be used anywhere a string would be used, most often signified by single or double quotes.

For example, the following would initialize the HOST argument with the value localhost, and will be used in the creation of the ros-bridge:

```
<arg name='host' default='localhost'/>

<include file="$(find carla_ros_bridge)/launch/carla_ros_b
ridge.launch">

    <arg name='host'

        value='$(arg host)'/>

...
```

In essence, using the $ identifier specifies that the immediate portion of the string encapsulated in parentheses should contain a command and value pair. For arguments, the command is arg and the value is the name of the argument intended to be used. If the argument is not found in the file above the current line or does not have a value set for it (which can be prevented by setting a default value), the launch file will throw an error, and will shutdown any ROS nodes created by the file.

In order to boot the ros-bridge with the launch file, an include child pointing to the ros-bridge launch file developed by CARLA must exist inside the launch file, such as:

```
<!--    BOOT ROS <-> CARLA INTERFACE  -->
    <include
                file="$(find carla_ros_bridge)/launch/carla_ros_
bridge.launch">
                <arg name='host'
                                value='$(arg host)'/>
                <arg name='port'
                                value='$(arg port)'/>
    </include>
```

As noted previously, two arguments were stored: host and port. By storing this information as arguments, it allows the ROS package to point to any CARLA instance running on any port or IP, including ones not located on the local machine.

SPAWN EGO-VEHICLE

The term ego-vehicle indicates a vehicle in the CARLA simulation that is intended to be interfaced with using the ros-bridge. As such, any vehicle named ego-vehicle (or other custom names if a custom config file is loaded) will have publisher and subscriber topics created so that sensor information can be read and vehicle commands can be sent.

Spawning an ego-vehicle is done in a similar manner to booting ros-bridge, however an additional file is needed in order to communicate vehicle and sensor information. A JSON file is created and passed to a CARLA developed launch file indicated which vehicles the ego-vehicle can choose from (ranging from an exact model to vehicle size description).

JSON is a data structure designed to easily represent key value pairs in a hierarchical manner, allowing for definitions of parent-child relationships.

This specific JSON file uses the following format:

```
{
    "sensors": [
                {
                                SENSOR 1
                        },
                        {
                                SENSOR 2
                        }
                ]
}
```

The outer curly braces indicated the vehicle parent, that has a list of child sensors in an unordered list. Each sensor can be described with the following structure:

```
{
                type": "sensor.camera.rgb",
                                "id": "front",
                                "x": 2.0, "y": 0.0,
                                "z": 2.0, "roll": 0.0,
                                "pitch": 0.0, "yaw": 0.0,
                                ...
}
```

Each field shown is mandatory for every sensor to successfully initialize. Type is a String matching one of the eight identifiers for each sensor supported in CARLA, while id is the specific reference identifier referenced in code to retrieve sensor data. X, Y, Z specify the sensors position relative to the vehicles center of geometry, or the mean of all X, Y, and Z points in the model. Furthermore, Positive X moves towards the front of the vehicle, Positive Z moves towards the Roof of the vehicle, and Positive Y moves towards the right side of the vehicle. Finally roll pitch and yaw represent the angles from the sensor origin for each dimension.

Finally, each sensor may require additional information in order for it to be properly initialized, such as camera image size. Other properties can be optional, with included defaults if not specified in the JSON file. All of these specifics can be found on the CARLA documentation for cameras and sensors [14].

Any number of supported sensors may be strung together inside the sensors list and will each create sensor topic information as denoted in the ros-bridge github repository. In the launch file, another $ denoted argument will be used to indicate the location of the sensor information JSON file:

```
<arg name="sensor_definition_file"
default="$(find autonomous_server)/config/sensors.json"/
>
```

The find command utilizes a ROS feature that, if a package is successfully compiled, and the devel/setup.bash for the specific workspace has been sourced, indexes the location of the package in a global list. This has the benefit of allowing named references to packages return full path information, allowing for development to be system independent.

In essence, the find command will lookup the specified package name from the global list of packages, and return the full system path to its root directory, allowing for files inside the package to be directly referenced.

Like the ros-bridge initialization, the ego vehicle will be created by including the CARLA developed carla_ego_vehicle.launch file from within the carla_ego_vehicle package. This launch file includes argments to identify a string to be passed to the vehicle filter, the sensors JSON file created previously, and the name of the vehicle, often defaulted to ego_vehicle.

With these two files included, the current launch file will connect to the CARLA instance, spawn a vehicle from the vehicles matching the filter, equip it with a sensor array defined by the sensors JSON file, and start publishing topics for each sensor, as well as general vehicle information, world information, and diagnostic information.

CONTROL THE VEHICLE

Out of the box the carla-ros-bridge supports 3 methods of commanding the ego-vehicle to move: autopilot, throttle / steer, and Ackermann control.

Autopilot is a useful command for testing, and simulating semi-random pedestrians and other outside vehicles. Each map has what is in essence a navigation mesh, or a set of pre-programmed lines following each road lane. When a vehicles CARLA based autopilot is enabled, the vehicle simply picks a random location it can traverse to on the map, and then follows the navigation mesh to its destination. It will also stop for vehicles in its path, stop at stop lights and stop signs, and wait for a left turn to be safe before taking it.

Another method is the throttle / steer method, whereby each axis of movement is set to a specific percentage. Steering can be anywhere from 100% left (written as -1.0), to centered (written as 0.0), or 100% right (written as 1.0). Likewise, throttle and brake are set as a value between 0 and 1, fully off and fully on respectively. Vehicle gear can also be set to an integer from 1 to n, n being the highest gear available in the specific ego vehicle. And finally, a Boolean variable represents whether the handbrake is being applied or the vehicle is in a reverse gear. This model is the backbone for all other control models, as even the Ackermann model provided by CARLA translates into the throttle / steer model.

Finally, there exists a pre-built Ackermann control module, utilizing a simple PID algorithm, which is very useful for developing an autonomous vehicle in CARLA intended to be used in the real world. Using the ROS AckermannDrive message, the navigation stack only calculates a desired forward velocity, acceleration, jerk, and desired wheel angle and angle velocity. Utilizing formulas found from an article written by Jarrod Snider [15], vehicle velocity and wheel angle can easily be calculated given a path and the vehicles relative position to the path.

One Caveat using the Ackermann controller is that it needs to also be initialized alongside the ros-bridge. Doing so is very similar to the ros-bridge, by including the carla_ackermann_control.launch file inside the one of the packages launch files.

BASIC LIDAR PROCESSING WITH PCL

Point Cloud Library is the most popular point cloud processing solution for ROS packages, and as such has great support in forms of tutorials and internet help boards. It is illustrated in Figure 3. Keep in mind that, at the time of this paper, no official or feature complete version of PCL has been translated to python 2.7, and thus it is recommended that packages do all Point Cloud processing in C++ nodes.

The overall workflow of using PCL in ros is:

1. Subscribe to a sensor_msgs::PointCloud2 topic from a lidar sensor
2. Convert the sensor_msgs::PointCloud2 data to pcl::PCLPointCLoud2 data
3. Process the Point Cloud
4. Publish either the resultant Point Cloud as a sensor_msgs::PointCloud2 or other information that can inform the vehicle of its calculations

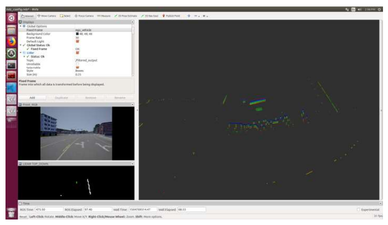

Figure 3. RVIZ Display showing filtered LIDAR data using a voxel grid filter to trim dense sections of the point cloud and a pass-through filter culling any points that won't collide with the vehicle in the vertical axis. A top down heatmap is generated in the bottom left.

The first step shall be left to the reader, as that process does not differ from subscribing to any other ROS topic in C++.

Inside the subscriber call back, to translate the sensor_msgs point cloud to the pcl point cloud, use:

```
void lidar_cb(const sensor_msgs::PointCloud2ConstPtr&
cloud_msg)

{

            pcl::PCLPointCloud2::Ptr cloud(new pcl::PCLP
ointCloud2);

            pcl_conversions::toPCL(*cloud_msg, *cloud);
```

the call back takes a reference to a pointer pointing to the data collected from the lidar sensor, and is converted to its pcl counterpart, and then stored inside the memory address pointed to by the pcl pointer named cloud.

After this is complete, any tutorial operating on pcl::PointCloud2 data can be followed, in this specific example a voxel grid filter will be used to trim any groups of points that are too close to one another. This will vastly lower the computational cost of working on the lidar data, while keeping its shape intact. This can be done by:

```
pcl::PCLPointCloud2::Ptr cloud_voxel(new pcl::PCLPointCloud2);

//voxel grid filter

pcl::VoxelGrid<pcl::PCLPointCloud2> sor;

sor.setInputCloud(cloud);

sor.setLeafSize(0.25, 0.25, 0.25);

sor.filter (*cloud_voxel);
```

A pointer is created to point to the resultant data. A voxelGrid filter is then created, given the converted point cloud as its input, given a leaf size, then computed, returning its results in the pointer created.

This is the basic framework with which point clouds are manipulated in PCL: initialize a pointer and a filter, set its input and variables, and then compute the result and store it in the memory location pointed to by the pointer. A chain of these can also be created, where the pointer of the first result becomes the input for the second filter, and so on.

Once all work on the lidar data has been completed, it can often be useful to publish its filtered output for use in other nodes. This is done by once again converting the data back into its original form, and then published on a separate topic, like so:

```
sensor_msgs::PointCloud2 output;

pcl_conversions::fromPCL(*cloud_plane, output);
```

REFERENCES

[1] Dosovitskiy A., Ros, G., Codevilla, F., Lopez A. and Koltun, V., 2019, "CARLA: An Open Urban Driving Simulator," Proceedings of the 1st Annual Conference on Robot Learning, pp. 1-16.
[2] Muller M., Dosovitskiy A., Ros, G., Codevilla, F., Lopez A. and Koltun, V., 2019, "CARLA: An Open Urban Driving Simulator," Proceedings of the 1st Annual Conference on Robot Learning, pp. 1-16.
[3] Dosovitskiy A., Ros, G., Codevilla, F., Lopez A. and Koltun, V., 2019, "CARLA Documentation." from https://carla.readthedocs.io/en/latest/
[4] Canonical Ltd., 2018, "Ubuntu 16.04.6 LTS (Xenial Xerus)." from http://releases.ubuntu.com/16.04/
[5] Balena, 2019, "Balena Etcher." from https://www.balena.io/etcher/
[6] Canonical Ltd., 2018, "Create a bootable USB stick on macOS." from https://tutorials.ubuntu.com/tutorial/tutorial-create-a-usb-stick-on-macos#0
[7] Open Source Robotics Foundation, 2017, "Ubuntu Install of ROS Kinetic." from http://wiki.ros.org/kinetic/Installation/Ubuntu
[8] Dosovitskiy A., Ros, G., Codevilla, F., Lopez A. and Koltun, V., 2019, "Getting Started with CARLA." from https://carla.readthedocs.io/en/latest/getting_started/
[9] CARLA, 2019, "CARLA Simulator" from https://github.com/carla-simulator/carla
[10] CARLA, 2019, "ROS bridge for CARLA simulator" from https://github.com/carla-simulator/ros-bridge
[11] zhanwenchen, 2018, "Installing CUDA kit 9.0 for Ubuntu 16.04.4 LTS" from https://github.com/carla-simulator/ros-bridge
[12] eric-wieser, 201, "ros_numpy" from https://github.com/eric-wieser/ros_numpy
[13] Open Source Robotics Foundation, 2015, "pcl" from http://wiki.ros.org/pcl
[14] Dosovitskiy A., Ros, G., Codevilla, F., Lopez A. and Koltun, V., 2019, "CARLA Documentation." from https://carla.readthedocs.io/en/stable/cameras_and_sensors/
[15] Jarrod M. Snider, 2009, "Automatic Steering Methods for Autonomous Automobile Path Tracking," CMU-RI-TR-09-08, Robotics Institute, Carnegie Mellow University, Pittsburgh, Pennsylvania