BUILDING SELF-DRIVING CAR ARCHITECTURE WITH ROBOT OPERATING SYSTEM

ISsoft Insights 2019 – Workshop Guide

Abstract

In this document you find a detailed guidance about how to build a self-driving car architecture with Robot Operating System

Alexey Sas, Oleg Karasik

alexey.sas@coherentsolutions.com, oleg.karasik@coherentsolutions.com

1 INTRODUCTION

Welcome to **ISsoft Insights 2019** workshop **“Building Self-Driving Car architecture with Robot Operating System”**.

This “tiny” document contains step-by-step guidance about how to develop a simple yet working self-driving car application.

To make workshop more interactive we have decided to remove all overhead related to initial environment configuration and setup. For each attendee we have created a dedicated virtual environment in Azure with preconfigured software.

All required materials including source code, detailed information about installed software and its configuration is available here

Stay focused and have fun! Let’s go!

2 DOCUMENT LEGEND

Besides text you will see multiple types of blocks in this document:

Command Snippet
This block contains a command prompt command (or several commands, each in a separate row). If not mentioned explicitly you should copy this command to the last used command prompt – regardless whether this is **Anaconda Prompt**, **Windows Command Prompt** or **Bash Shell**. In case of multiple commands – each command should be copied and executed in order of appearance.

COMMAND SNIPPET
`<command>`

Code Snippet
This block contains file name, action, relative position and piece of code block to insert. If information is omitted, then specified content of the file should be replaced with code block.

CODE SNIPPET
`<file> (action, relative position)`
`<code>`

This block also has a second variation used to express replacement of one code block with another.

CODE SNIPPET
`<file> (replace, relative position)`
`<original code>`
`with`
`<replacement code>`

Attention
This block contains important information or comment. While block content does not directly affect the step, where it is mentioned, its content is worth reading.

`<important notice or warning>`

3 PHASE 1: PREPARATIONS

3.1 CONNECT TO REMOTE ENVIRONMENT

The first step in our workshop is to make connection to remote virtual environment. You can do this from both Windows or Mac.

On Windows

Use Remote Desktop Connection application to connect to remote environment.

On Mac

Use Microsoft Remote Desktop 10 application to connect to remote environment

Place connection details here

When connected you should see standard Windows 10 desktop.

3.2 START BACKGROUND SOFTWARE

While all software is preinstalled, it is not set to run automatically on system startup. Why? That is easy – running this software is a part of the development process we would like to get acquainted with.

3.2.1 START XSERVER

The first element we need to launch is the **XServer**. This application would create a **XServer Display** used by Linux subsystem to render graphics.

On the top left corner of Windows desktop there is a shortcut called **"XLaunch"**

Double click on it to run.

DO NOT change any application settings. Just click: **Next → Next → Next → Finish**.

3.2.2 START CARLA SERVER

CARLA is a complex self-driving car simulator based on Unreal Engine. In general, simulator consists of two parts: a client and a server. The server part is the "world" with building, traffic lights, pedestrians, cars, …, etc.

During the workshop we will use one of the default "worlds" – a city map created by CARLA team.

To start CARLA server, we need to open a new **Anaconda Prompt**. In the search bar (on the bottom left) type "**anaconda**" and open "**Anaconda Prompt**" from best matches.

In the opened command prompt type:

COMMAND SNIPPET

```
cd C:\Workshop\carla-server
```

- This sets current directory of anaconda prompt to: **C:\Workshop\carla-server**

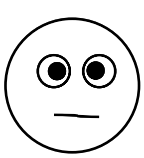
Launch the CARLA Server using the following command:

COMMAND SNIPPET

```
CarlaUE4.exe -windowed -ResX=640 -ResY=480
```

- This launches CARLA server in windowed mode with windows size set to 640x480 pixels.

Application could take a while to start but when it is done you should see a picture like this:



When CARLA Server is started it will capture your cursor inside it. Use **ALT+TAB** to switch out and **minimize** the window.

DO NOT CLOSE CARLA SERVER WINDOW

3.2.3 START CARLA – ROS BRIDGE

To enable interactions between ROS and CARLA server we need to establish a "bridge" between them. CARLA ROS bridge is a ROS node created with knowledge of CARLA server API. Bridge defines a set of messages and topics other nodes can use to send commands to CARLA server and receive information about what is happening in the world.

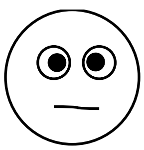
To start the bridge, we need to open a new **bash command prompt**. In the search bar type "**bash**" and open "**Bash**" from best matches.

In the opened command prompt type:

COMMAND SNIPPET

```
roslaunch carla_ros_bridge client.launch
```

- This launches ROS master process (covered later) and **carla_ros_bridge** node.



DO NOT close bash prompt - **minimize** it.

3.2.4 START CARLA CLIENT

To control a vehicle, we need… a vehicle. We will not use custom code to create a vehicle, but instead of this we would use client created by CARLA team. The client creates a random vehicle, initializes its sensors and allows you to manually control it by using keyboard.

To start client, we need to open new **Anaconda Prompt**. In the opened command prompt type:

COMMAND SNIPPET

```
cd C:\Workshop\carla-client
```

```
python manual_control.py --res 640x480
```

- This launches **manual_control.py** python script in a window of 640x480 pixels.

When client is launched you should see a picture from car's camera.

You can check anaconda prompt for instruction of how to control the vehicle and play with it for a while.

When you are done **close** the window and **relaunch** the client.

DO NOT close newly opened window - **minimize** it.

3.3 Creating ROS package

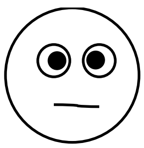
ROS package contains code, scripts, message and files required to build and make nodes from this package discoverable. We would start from the simplest scenario – an empty package.

Open a new **bash prompt** and type the following commands:

COMMAND SNIPPET

```
mkdir -p /mnt/c/Workshop/sdc_workspace/catkin_ws/src
cd /mnt/c/Workshop/sdc_workspace/catkin_ws
```

- This creates **/mnt/c/Workshop/sdc_workspace/catkin_ws/src** and sets bash prompt current directory to **/mnt/c/Workshop/sdc_workspace/catkin_ws**.



In Windows subsystem for Linux path **/mnt/*** is mapped to appropriate logical drive i.e. **/mnt/c** → **C:**, **/mnt/d** → **D:**.

The directories we have just created are required to initialize **catkin workspace**. You can think of Catkin as of build infrastructure used to create ROS packages – very similar to **angular-cli** for Angular.

This command will initialize a new **catkin workspace**:

COMMAND SNIPPET

```
catkin_make
```

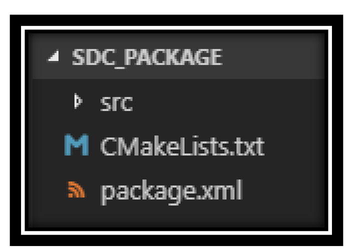
Package is also created using Catkin:

COMMAND SNIPPET

```
cd src
catkin_create_pkg sdc_package sensor_msgs cv_bridge std_msgs rospy
```

- This invokes **catkin_create_pkg** macro which creates a new package named **sdc_package** which depends on: **sensor_msgs**, **cv_bridge**, **std_msgs** and **rospy** packages.

Using **File Explorer** navigate to **C:\Workshop\sdc_workspace\catkin_ws\src\sdc_package** and open it in **Visual Studio Code**.



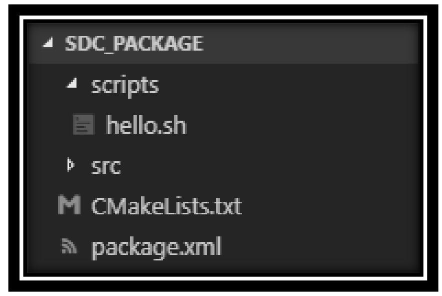
4 PHASE 2: PROGRAMMING ROS NODES

4.1 CREATE "HELLO WORLD" NODE

The first thing to begin with is always a "Hello World" something. In case of ROS – "Hello World" node.

In **Visual Studio Code** create new **scripts** directory inside **sdc_package** directory.

Then inside it, create **hello.sh** file.



Inside the **hello.sh** type the following code:

CODE SNIPPET

scripts/hello.sh

```
#!/bin/bash
echo Hello World!
```

- Line #1 instructs shell to execute this as **bash shell** script.
- Line #2 uses **echo** application to print "Hello World!" into terminal.

We already have ROS master process online (we have launched it when started CARLA ROS bridge) so to run **hello.sh** node all we need is to open the **bash command prompt** we have used to create package and execute the following commands:

COMMAND SNIPPET

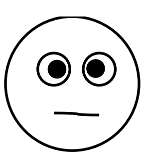
```
cd ..
```

```
source devel/setup.bash
```

```
rosrun sdc_package hello.sh
```

- #1 – changes current directory to **/mnt/c/Workshop/sdc_workspace/catkin_ws**.
- #2 – loads workspace related environment variables into shell context.
- #3 – runs **hello.sh** node from **sdc_package**.

As the result you should see **"Hello World"** output in terminal.



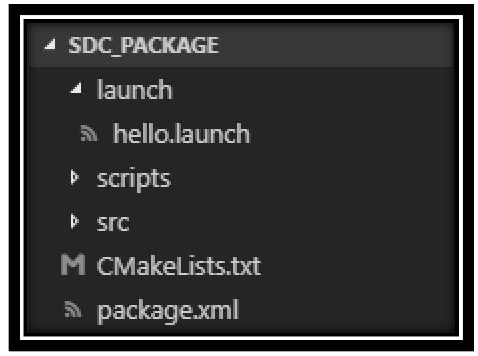
Future reference
You can start ROS master process using **roscore** command in **bash command prompt.**

Running nodes like this could be fine when you have one or two nodes but when there are multiple nodes and these nodes have its own configuration running all this setup manually could be problematic.

This problem is solved by the infrastructure. It allows you to define a **.launch** with list of nodes, parameters and references to another **.launch** files and launch this with single command.

In **Visual Studio Code** create new **launch** directory inside **sdc_package** directory.

Then inside it, create **hello.launch** file.



Inside the **hello.launch** type the following code:

CODE SNIPPET

launch/hello.launch

```
<?xml version="1.0"?>
<launch>
  <!--
    pkg: defines package name
    type: defines file to launch
    name: defines unique node name
    output: defines application output destination
  -->
  <node
    pkg="sdc_package"
    type="hello.sh"
    name="hello_node"
    output="screen">
  </node>
</launch>
```

Open the **bash command prompt** we have previously used to run **hello.sh** and type the following command:

COMMAND SNIPPET

```
roslaunch sdc_package hello.launch
```

- This starts **roslaunch** server to connect to ROS master process and run all ROS nodes included in **.launch** file.

As the result you should see **"Hello World"** output in terminal.

4.2 Create camera node

We have already created node to execute the **.sh** script but it had no logic and literally did nothing except printing “Hello World!”. It is time to write something that has logic behind it. In our case we write a node to capture, process and display images from car’s camera.

Open File Explorer in **C:\Workshop\project\templates** directory and copy **camera.py** and **utilities.py** to **sdc_package\scripts** directory.

Open **scripts\camera.py** in **Visual Studio Code** and configure file to be executed as **python script**.

CODE SNIPPET

scripts/camera.py (insert after, TODO: 1)

```
#!/usr/bin/env python
```

All ROS nodes should have a unique name. In python scripts this is done by using the following code:

CODE SNIPPET

scripts/camera.py (insert after, TODO: 2)

```
rospy.init_node('camera')
```

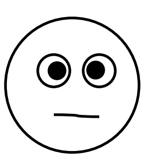
Now we need to do one of the essential things in ROS – we would subscribe to a topic. In our case this would be one of the topics created by CARLA ROS bridge. This topic supplies images from car’s front camera.

CODE SNIPPET

scripts/camera.py (insert after, TODO: 3)

```
rospy.Subscriber(
  '/carla/ego_vehicle/camera/rgb/front/image_color',
  Image,
  self.process_image)
```

- This subscribes **self.process_image** to run every time **'/carla/ego_vehicle/camera/rgb/front/image_color'** topic has new **Image** message.



The **‘Image’** message format description can be found here or by QR Code

The last edit in **camera.py** is to implement the callback:

CODE SNIPPET

scripts/camera.py (insert after, TODO: 4)

```
def process_image(self, image):
  self.got_image = True
  self.image = self.bridge.imgmsg_to_cv2(image, "bgr8")
```

We have done with the script and now we need to create **camera.launch** in **launch** directory.

In **launch** directory create **camera.launch** with the following content:

CODE SNIPPET

launch/camera.launch

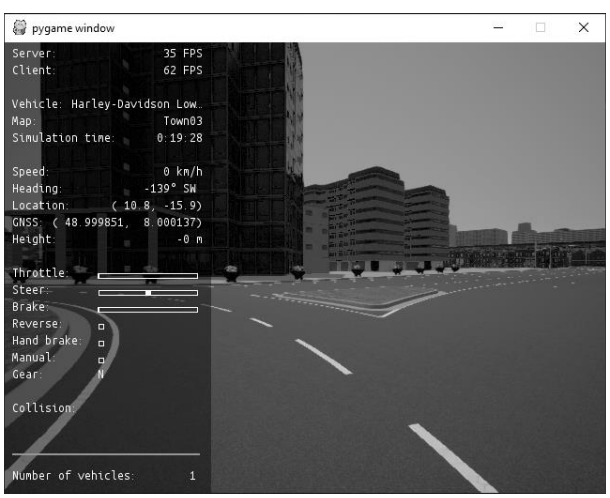
```
<?xml version="1.0"?>
<launch>
  <node
    pkg="sdc_package"
    type="camera.py"
    name="camera"
    output="screen">
  </node>
</launch>
```

Open the **bash command prompt** we have previously used to run **hello.launch** and type the following command:

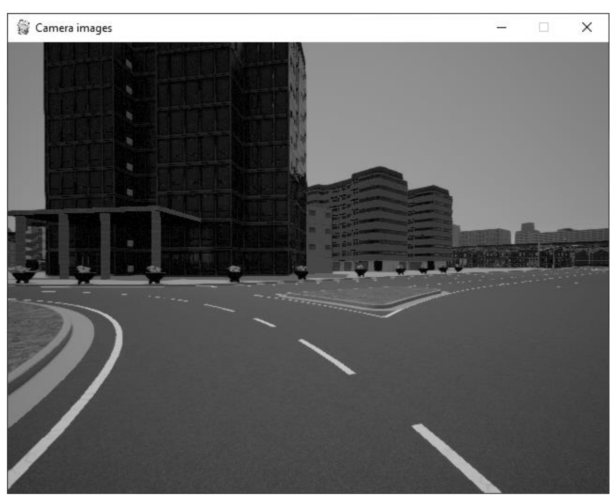
COMMAND SNIPPET

```
roslaunch sdc_package camera.launch
```

As the result you should see the picture from car's front camera:



View from **manual_control.py**



View from **camera.launch**

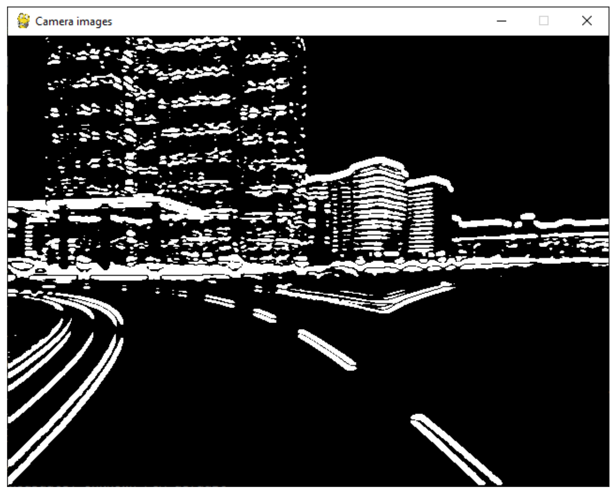
Sounds boring? Then let's add more fun.

Do the following change in **camera.py** and **relaunch** camera node:

CODE SNIPPET
`scripts/camera.py (uncomment, TODO: 5)`
`# frame = pipline(frame)`

Now it looks cool! The **pipeline** function we uncommented filters image to make it useful for line detection.

View from **manual_control.py**



View from **camera.launch**

4.3 Create controller node

Capturing camera images allows us to see what is happening with our vehicle. In more realistic scenarios we could make further image processing to extract more data, but this is more about mathematics and algorithms rather than self-driving cars.

Our next step is to start controlling the vehicle. This can be done by publishing a **CarlaVehicleControl** message to **'/carla/ego_vehicle/vehicle_control_cmd'** topic.

Create a new **msg** directory in **sdc_packages** directory. Inside **msg** directory create **CarlaVehicleControl.msg**.

Inside **CarlaVehicleControl.msg** type the following code:

CODE SNIPPET
msg/CarlaVehicleControl.msg

```
Header header
float32 throttle
float32 steer
float32 brake
bool hand_brake
bool reverse
```

Simply adding a message into a package is not enough. The reason behind it is very simple – the message is the **binary** contract. So, to make it work we need to **compile** all messages included into our package. Package build is defined in two files: **package.xml** and **CMakeList.xml**.

Define **sdc_package** dependency on **message_generation** and **message_runtime** packages:

CODE SNIPPET
package.xml (insert at, line: 56, 65)

```
<build_depend>message_generation</build_depend>
```

```
<exec_depend>message_runtime</exec_depend>
```

Include **message_generation** package into list of packages to find on build **(find_package)**:

CODE SNIPPET
CMakeList.txt (insert at, line: 15)

```
message_generation
```

Include **message_runtime** into dependencies list of **sdc_package** library:

CODE SNIPPET
CMakeList.txt (replace, lines: 107-112)

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES sdc_package
#  CATKIN_DEPENDS cv_bridge rospy sensor_msgs std_msgs
#  DEPENDS system_lib
)
```

with

```
catkin_package(
  LIBRARIES sdc_package
  CATKIN_DEPENDS cv_bridge rospy sensor_msgs std_msgs message_runtime
)
```

Include **CarlaVehicleControl.msg** into list of messages to compile for our package:

CODE SNIPPET

CMakeList.txt (replace, lines: 52-56)

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

with

```
add_message_files(
  FILES
  CarlaVehicleControl.msg
)
```

Turn on message generation build step:

CODE SNIPPET

CMakeList.txt (replace, lines: 72-75)

```
# generate_messages(
#   DEPENDENCIES
#   sensor_msgs#   std_msgs
# )
```

with

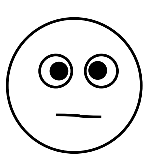
```
generate_messages(
  DEPENDENCIES
  sensor_msgs
  std_msgs
)
```

Open the **bash command prompt** we have previously used to run **camera.launch** and type the following commands:

COMMAND SNIPPET

```
catkin_make
```

```
source devel/setup.bash
```



Probably you've noticed that **bash** prompt you've opened is busy with ROS running our **camera.launch.** To stop ROS, you should use **CTRL+C** combination – this would trigger ROS termination.

Messages compilation can be verified with the following command:

COMMAND SNIPPET

```
rosmsg show sdc/CarlaVehicleControl.msg
```

Copy **simple_controller.py** from **templates** directory to **sdc_package\scripts** directory and open it in **Visual Studio Code**.

The **simple_controller.py** is a template we would later tweak to implement final controller but no worries it is prepared for simple implementation too.

Include **CarlaVehicleControl** message:

CODE SNIPPET

scripts/simple_controller.py (insert after, TODO 1.1)

```
from sdc_package.msg import CarlaVehicleControl
```

Initialize node name:

CODE SNIPPET

scripts/simple_controller.py (insert after, TODO 1.2)

```
rospy.init_node('simple_controller')
```

Create publisher to publish messages to **'/carla/ego_vehicle/vehicle_control_cmd'** topic:

CODE SNIPPET

scripts/simple_controller.py (insert after, TODO: 1.3)

```
self.control_publisher = rospy.Publisher(
  '/carla/ego_vehicle/vehicle_control_cmd',
  CarlaVehicleControl,
  queue_size = 1
)
```

Enable basic control routine:

CODE SNIPPET

scripts/simple_controller.py (uncomment, TODO: 1.4)

```
#throttle_output, steer_output, brake_output = self.basic_control()
```

Prepare and publish **CarlaVehicleControl** message:

CODE SNIPPET

scripts/simple_controller.py (insert after, TODO: 1.5)

```
message = CarlaVehicleControl(
  None, throttle_output, steer_output, brake_output, False, False)

self.control_publisher.publish(message)
```

In **launch** directory create **simple_controller.launch** with the following content:

CODE SNIPPET

launch/simple_controller.launch

```
<?xml version="1.0"?>
<launch>
  <node
    pkg="sdc_package"
    type="simple_controller.py"
    name="simple_controller"
    output="screen">
  </node>
</launch>
```

Running **simple_controller** node without visualization does not make much sense. That is why it is time to create a launch file of launch files!

Create **all.launch** file in **launch** directory with the following content:

CODE SNIPPET

launch/all.launch

```
<?xml version="1.0"?>
<launch>
  <include file="$(find sdc_package)/launch/simple_controller.launch" />
  <include file="$(find sdc_package)/launch/camera.launch" />
</launch>
```

Open the **bash command prompt** we have previously used to run **catkin_launch** and launch **all.launch**:

COMMAND SNIPPET

```
roslaunch sdc_package all.launch
```

As the result you should see how your vehicle is slowly moving forward.

5 PHASE 3: COMPLETE SELF-DRIVING CAR ARCHITECTURE

Real self-driving car controller requires minimum two sources of information. The first one is the mission route and the second one is a close-range local route. These two sources of information would be implemented using two additional nodes: **mission planner** and **local planner**.

5.1 DEFINING DATA AND MESSAGES

Create **data** directory in **sdc_package** directory and copy **waypoints.csv** from **C:\Workshop\project\data** directory.

In **msg** directory create new messages – **BaseWaypoint.msg**, **Path.msg**, **Waypoint.msg** and **LocalPath.msg** with the following content:

CODE SNIPPET

msg/BaseWaypoint.msg

```
float32 x
float32 y
```

msg/Path.msg

```
BaseWaypoint[] waypoints
```

msg/Waypoint.msg

```
float32 x
float32 y
float32 v
```

msg/LocalPath.msg

```
Waypoint[] waypoints
```

In **CMakeLists.txt** add these messages to the end of **add_message_files()** block:

CODE SNIPPET

CMakeLists.txt (insert at, line: 55)

```
BaseWaypoint.msg
Path.msg
Waypoint.msg
LocalPath.msg
```

Open the **bash command prompt** we have previously used to run **all.launch** and type the following command:

COMMAND SNIPPET

```
catkin_make
```

```
source devel/setup.bash
```

5.2 CREATE MISSION PLANNER NODE

Copy **mission_planner.py** from **templates** directory to **sdc_package/scripts** directory and open it in Visual Studio Code.

Include **BaseWaypoint** and **Path** messages:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 1)

```
from sdc_package.msg import BaseWaypoint, Path
```

Initialize node name:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 2)

```
rospy.init_node('mission_planner')
```

We create publisher that will be used to publish **Path** message to **'/planner/mission_waypoints' topic**:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 3)

```
self.waypoints_publisher = rospy.Publisher(
  '/planner/mission_waypoints',
  Path,
  queue_size = 1)
```

We will not retrieve mission waypoint from real source (GPS, road detection, …, etc.). Instead of this we will load and publish a predefined set of endpoints.

Firstly, we should obtain a path to **waypoints.csv** data file:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 4)

```
waypoints_file_path = rospy.get_param('~waypoints_path')
```

Secondly, we will load waypoints by parsing input CSV into array of **BaseWaypoint** objects by calling **self.load_waypoints** function:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 5)

```
waypoints = self.load_waypoints(waypoints_file_path)
```

Thirdly, we will publish loaded waypoints by calling **self.publish_waypoints** function:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 6)

```
self.publish_waypoints(waypoints)
```

Fourthly, we will put our node into empty loop by calling **rospy.spin()** function:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 7)

```
rospy.spin()
```

Now we need to tweak **self.load_waypoints** and **self.publish_waypoints** functions.

In **self.load_waypoints** we need to implement CSV parsing logic (which is quite simple):

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 8)

```
waypoint = BaseWaypoint()
waypoint.x = float(row[0])
waypoint.y = -float(row[1])
waypoints.append(waypoint)
```

In **self.publish_waypoints** we need to create new **Path** object from input array of **BaseWaypoint** objects:

CODE SNIPPET

scripts/mission_planner.py (insert after, TODO: 9)

```
path = Path(waypoints)
```

The last thing left is to create a **mission_planner.launch** file and include it into **all.launch**. In **launch** directory create **mission_planner.launch** with the following content:

CODE SNIPPET

launch/mission_planner.launch

```
<?xml version="1.0"?>
<launch>
  <node
    pkg="sdc_package"
    type="mission_planner.py"
    name="mission_planner"
    output="screen">
    <param name="waypoints_path" value="$(find sdc_package)/data/waypoints.csv" />
  </node>
</launch>
```

In **all.launch** include **mission_planner.launch**:

CODE SNIPPET

launch/all.launch (insert at, line: 3)

```
<include file="$(find sdc_package)/launch/mission_planner.launch" />
```

5.3 CREATE LOCAL PLANNER NODE

Copy **local_planner.py** from **templates** directory to **sdc_package** directory and open it in Visual Studio Code.

Include **BaseWaypoint**, **Waypoint**, **Path** and **LocalPath** messages:

CODE SNIPPET

scripts/local_planner.py (insert after, TODO: 1)

```
from sdc_package.msg import BaseWaypoint, Waypoint, Path, LocalPath
```

Initialize node name:

CODE SNIPPET

scripts/local_planner.py (insert after, TODO: 2)

```
rospy.init_node('local_planner')
```

Subscribe on **'/planner/mission_waypoints'** topic to receive mission path:

CODE SNIPPET

scripts/local_planner.py (insert after, TODO: 3)

```
rospy.Subscriber(
  '/planner/mission_waypoints',
  Path,
  self.init_mission)
```

Now we need to tweak **self.process_position**, **self.plan_path**, **self.get_car_closest_waypoint_idx** and **self.prepare_local_path** functions.

The **self.process_position** function is a callback function registered to receive **Odometry** messages. **Odometry** is a standard ROS message type that contains information about current vehicle position, orientation, speed, …, etc.

In this guide we are interested in three components:

- **odometry.pose.pose.position** – to get current vehicle position
- **odometry.pose.pose.orientation** – to get current vehicle orientation
- **odometry.twist.twist.linear** – to get current vehicle speed

> The ‘Odometry’ message format description can be found here or by QR Code

In **self.process_position** function store position component of **Odometry** message in **self.position**:

CODE SNIPPET
`scripts/local_planner.py (insert after, TODO: 4)`

```
self.position = position.pose
```

In **self.plan_path** function we need to publish **LocalPath** create by **self.prepare_local_path**:

CODE SNIPPET
`scripts/local_planner.py (insert after, TODO: 5)`

```
self.local_publisher.publish(local_path)
```

In **self.get_car_closest_waypoint_idx** function we need to find index of closest waypoint to current vehicle position:

CODE SNIPPET
`scripts/local_planner.py (insert after, TODO: 6)`

```
closest_idx = self.waypoint_tree.query([x, y], 1)[1]
```

Now we have closest waypoint index and can get waypoints subset from the stored waypoints started from this index.

Not exactly true, if we have closest waypoint which is already behind the vehicle we cause vehicle to make a 180-degree turn. So, we can add additional check to see if closest waypoint behind and if so – take next waypoint.

From math the easiest way to check this is to use sign of the dot product of vectors.

> Graphical explanation of usage of Sign of Dot Product to make sure A and B facing similar direction can be found here or by QR Code

<u>In total we need:</u>

1. Get next point.
2. Calculate vectors based on closest waypoint and next waypoint.
3. Check if position and next waypoint are placed in one direction from closest waypoint. If so – select next waypoint as our reference waypoint for trajectory as closest waypoint is already behind.
4. Compare dot value with zero and select appropriate index.

CODE SNIPPET

scripts/local_planner.py (insert after, TODO: 7)

```
closest_waypoint = self.waypoints_xy[closest_idx]
next_waypoint = self.waypoints_xy[closest_idx + 1]

# convert waypoints to numpy vectors
closest_as_vector = np.array(closest_waypoint)
next_as_vector = np.array(next_waypoint)
position_vector = np.array([x, y])

# find vectors of interest
v1 =  next_as_vector - closest_as_vector
v2 =  position_vector - closest_as_vector

# calculate dot product
val = np.dot(v1, v2)

if val > 0:
    closest_idx = closest_idx + 1
```

In **self.prepare_local_path** function we need create **LocalPath** object by adding speed component to **BaseWaypoint** objects:

CODE SNIPPET

scripts/local_planner.py (insert after, TODO: 8)

```
result_waypoints = []

for base_waypoint in base_waypoints:
    waypoint = Waypoint()
    waypoint.x = base_waypoint.x
    waypoint.y = base_waypoint.y
    waypoint.v = PLANNED_SPEED

    result_waypoints.append(waypoint)

path = LocalPath()
path.waypoints = result_waypoints
```

The last thing left is to create a **local_planner.launch** file and include it into **all.launch**.

In **launch** directory create **local_planner.launch** with the following content:

CODE SNIPPET

launch/local_planner.launch

```
<?xml version="1.0"?>
<launch>
  <node
    pkg="sdc_package"
    type="local_planner.py"
    name="local_planner"
    output="screen">
  </node>
</launch>
```

In **all.launch** include **mission_planner.launch**:

CODE SNIPPET

launch/all.launch (insert at, line: 3)

```
<include file="$(find sdc_package)/launch/local_planner.launch" />
```

5.4 TWEAKING CONTROLLER NODE

The last part of this guide is about tweaking previously written controller to drive the vehicle down mission path.

Copy **pid.py** and **stanley.py** from **templates** directory to **sdc_package\scripts** directory.

Open **simple_controller.py** in **Visual Studio Code**.

Uncomment imports of **PID** and **Stanley** controllers:

CODE SNIPPET

scripts/simple_controller.py (uncomment, lines: 5,9-10)

```
#from pid import PID
#from stanley import Stanley
```

Import **LocalPath** and **Waypoint** messages:

CODE SNIPPET

scripts/simple_controller.py (insert after, TODO: 2.1)

```
from sdc_package.msg import LocalPath, Waypoint
```

Initialize **PID** and **Stanley** controllers by calling **self.init_controllers** function:

CODE SNIPPET
`scripts/simple_controller.py (insert after, TODO: 2.2)`
`self.init_controllers()`

Subscribe to **'/planner/local_waypoints'** to receive **LocalPath**:

CODE SNIPPET
`scripts/simple_controller.py (insert after, TODO: 2.3)`
`rospy.Subscriber('/planner/local_waypoints', LocalPath, self.process_waypoints)`

Subscribe to **'/carla/ego_vehicle/odometry'** to receive **Odometry**:

CODE SNIPPET
`scripts/simple_controller.py (insert after, TODO: 2.4)`
`rospy.Subscriber('/carla/ego_vehicle/odometry', Odometry, self.process_position)`

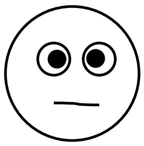
Uncomment call **self.wait_initialization()** to make sure we will wait for **LocalPath** and **Odometry** to arrive:

CODE SNIPPET
`scripts/simple_controller.py (uncomment, TODO: 2.5)`
`#self.wait_initialization()`

Now we need to tweak **self.process_position**, **self.process_waypoints**, **self.loop** and **self.control** functions.

The **self.process_position** function is a callback to receive **Odometry** messages. Inside this callback we need to store current vehicle position, velocity and heading direction. Current position is retrieved from the **Odometry** message itself. Heading direction is received using **quaternion**. We need to get Euler **yaw** angle from quaternion, so we can use this angle in Stanley controller (implemented in **self.get_yaw_last_position** function):

CODE SNIPPET
`scripts/simple_controller.py (replace after, TODO: 2.6)`
`self.x = None` `self.y = None`
`self.yaw = None`
`with`
`self.x = self.position.pose.position.x` `self.y = self.position.pose.position.y`
`self.yaw = self.get_yaw_last_position()`



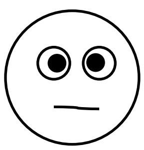
You can find a brief introduction to quaternion and Euler yaw angle [here](#) and [here](#) or by QR Codes

Quaternion

Euler Yaw Angle

Additionally, we have a separate x, y velocity components in the **Odometry** message. We can use the Pythagorean Theorem to calculate full velocity.

CODE SNIPPET
`scripts/simple_controller.py (replace after, TODO: 2.7)`
`self.v = None`
`with`
`self.v = math.sqrt(v_x * v_x + v_y * v_y)`



You can find Pythagorean Theorem description [here](#) or by QR Code

The **self.process_waypoints** function is a callback to receive **LocalPath** message. Inside this function we need to store waypoints we received from **local planner**:

CODE SNIPPET
`scripts/simple_controller.py (insert after, TODO: 2.8)`
`self.waypoints = path.waypoints`

Inside **self.loop** function we need to replace **throttle**, **steer** and **brake** parameters received from **self.basic_control** function with parameters received from **self.control** function:

CODE SNIPPET
`scripts/simple_controller.py (uncomment, TODO: 2.9)`
`#throttle_output, steer_output, brake_output =  self.control(dt)`

The controller is tweaked and as it is already linked to **all.launch** file there is nothing left!

6 PHASE 4: TEST DRIVE

We have implemented and tweaked everything we planned to and now it is time to test this beast!

Close and Reopen CARLA client.

Open the **bash command prompt** we have previously used to run **all.launch** and type the following command:

COMMAND SNIPPET

```
roslaunch sdc_package all.launch
```

Enjoy!

7 CONCLUSION

We have explored how to build a self-driving car architecture with Robot Operating System and CARLA self-driving car simulator.

While most of the time we were making "Find and Replace" operations rather than actual coding you still have the opportunity to explore all the mathematics behind this process by exploring links mentioned during the guide and extra information in **Extra** section.

Hope you have enjoyed the journey!

8 Extra

#	Title (Hyperlink)	QR Code
1	Workshop materials	
2	Automatic Steering Methods for Autonomous Automobile Path Tracking	
3	Graphical explanation of usage of Sign of Dot Product to make sure A and B facing similar direction	
4	Quaternion	
5	Euler Yaw Angle	
6	Pythagorean Theorem	