ECE 498SMA: Principles of safe autonomy Spring 2020

Written by: Yangge Li, Hebron Taylor, Joohyung Kim, Sayan Mitra *MP 2: Vehicle Model and Control*

Course Website *Due Date: 2020/2/21*

1 Introduction

In this MP, you will implement the vehicle model in the lecture and a waypoint tracking controller which can be important parts for the future MPs and projects. You will visualize the vehicle model you implemented in the Gazebo simulator and use the controller you implemented to control the vehicle to follow a series of waypoints. You will play around with the controller gains to achieve better performance of the controller. ROS is used in this MP to connect the vehicle model and controller to the simulator and acquire command from user.

The provided code can be found at git repo. All your code should be in file controller.py and you will have to write a brief report. In the MP you will get exposed to various different ROS messages which are widely used in ROS for communicating with or controlling the vehicle. This document gives you the first steps to get started on implementing the vehicle model and waypoint following controller. You will have to take advantage of tutorials and documentations available online. Cite all resources in your report. **All the regulations for academic integrity and plagiarism spelled out in the student code apply.**

Learning objectives

- Working with ros topics for vehicle control
- Programming executable vehicle models
- Programming waypoint following controller for vehicles

System requirements

- Ubuntu 16
- ROS Kinetic
- ros-kinetic-ros-control
- ros-kinetic-effort-controllers
- ros-kinetic-joint-state-controller
- ros-kinetic-ackermann-msgs

2 Module architecture

The following functions will be important for this MP. However, in this MP you only need to implement some of the functions. The functions marked by * are not *required* for you to implement, but you should read them and experiment with them.

getModelState* This function will call the "/gazebo/set_model_state" service that will return a message that contains the position, orientation, linear velocity and angular velocity of the vehicle. The return value of this function is a Gazebo message ModelState. This message is widely used in ROS describe a Gazebo model's pose, which includes position and orientation and twist, which includes linear and angular velocity. In this MP, we will mainly use the position and orientation information stored in the message. Note the orientation information is in form of quaternion in the message. The content in ModelState message **msg** can be accessed by

```
msg.pose.position.x
```

More details about ModelState message can be found here.

quaternion_to_euler* The function will take an orientation representated by quaternion and convert it to euler angle representation. The input to the function is the x, y, z, w component of orientation represented by quaternion and the output of the function is roll pitch and yaw component of orientation represented by euler angle. In this MP, we will focus on the yaw component of orientation.

rearWheelModel This function contains the mathematical model which will represent the vehicle. It will take in the output from controller as input and compute the rate of change of the position and orientation. The output from the controller is an AckermannDrive message. The AckermannDrive message is commonly used in ROS to drive car-like vehicle using AckermannDrive steering. In this MP, we will focus on the speed and steering_angle_velocity field of the AckermannDrive message. The content in AckermannDrive message **msg** can be accessed by

```
msg.speed
```

More details about AckermannDrive message can be found here.

rearWheelFeedback This function contains the controller which will enable the vehicle to drive to a waypoint. The function will take the current state (position and orientation) and the target state of the vehicle and use them to compute the speed and steering angle necessary to reach the waypoint. The current pose and target pose of the vehicle are stored in a ModelState message format.

The computed control input to the model will be packed into an AckermannDrive message and will be returned to be sent to the rearWheelModel function.

setModelState This function will set the new state of the vehicle to the Gazebo simulator. The function will take the current state and target state of the vehicle. It will first compute the controller input to the vehicle and then compute and update the state of the vehicle.

3 Development instructions

3.1 Running Gazebo Simulator

In this MP, you will work with the vehicle models in the Gazebo Simulator. To run the simulator, you should first go to the root directory of the files you downloaded from git repository where you should see a src folder. The next step is to run command

```
catkin_make
```

in the folder. There should be no error during the execution of the command and when finished, you should see two folders devel and build been generated.

The next step is to run command

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

in the root directory of the files you downloaded. This command should be executed every time before you try to run the simulator from a new terminal.

In order to see the building models in the environment you must execute the following command:

```
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:Path to src folder/src/mp2/models
```

After all the previous setup steps are finished, you can start the simulator by running command

```
roslaunch mp2 mp2.launch
```

You should be able to see the Gazebo simulator window open up and the vehicle in the simulator as shown in figure 1.

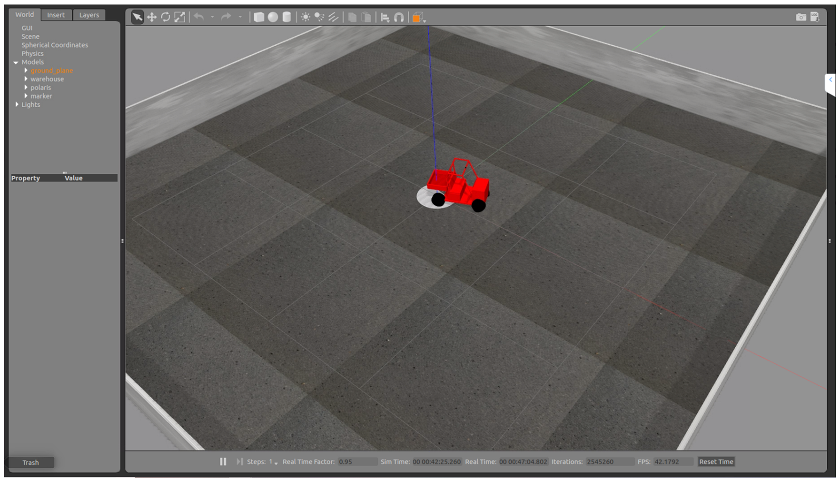

Figure 1: Gazebo simulator window

You can also use the command

```
roslaunch mp2 mp2_empty.launch
```

To launch the simulation without the building model

3.2 Implmenting the vehicle model

The vehicle model we used in this MP is the bicycle model we mentioned during the lecture. In this MP, we will focus on the rear wheel of the vehicle and therefore the dynamics of the model is given by

$$\dot{x_r} = v_r cos(\theta) \tag{1}$$
$$\dot{y_r} = v_r sin(\theta) \tag{2}$$
$$\dot{\theta} = \omega \tag{3}$$

The vehicle model should be implemented in function rearWheelModel. The procedure of this function should be:

1. Acquire the current state of the vehicle from Gazebo simulator
2. Convert the orientation of the vehicle from quaternion to euler angle
3. Compute the rate of change for the state variables by using the differential equations specified above
4. Return the computed result

3.3 Implementing the controller

In this part of the MP, a waypoint following controller will be implemented so that the vehicle can drive to a given waypoint. Given a target state x^* and gain constants k_1, k_2, $k_3 > 0$, define the control inputs at state x as follows:

$$v_r = (x^* \lceil v_r) cos(\theta_e) + k_1 x_e \tag{4}$$
$$\omega = x^* \lceil \omega + (x^* \lceil v_r)(k_2 y_e + k_3 sin(\theta_e)) \tag{5}$$

where the "error" vector $[x_e, y_e, \theta_e]$ at state x relative to the target state x^* is defined as:

$$\begin{bmatrix} x_e \\ y_e \\ \theta_e \end{bmatrix} = \begin{bmatrix} cos(x\lceil\theta) & sin(x\lceil\theta) & 0 \\ -sin(x\lceil\theta) & cos(x\lceil\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} (x^*\lceil x_r) - (x\lceil x_r) \\ (x^*\lceil y_r) - (x\lceil y_r) \\ (x^*\lceil \theta_r) - (x\lceil \theta_r) \end{bmatrix} \tag{6}$$

The controller should be implemented in function rearWheelFeedback. You should come up with the controller gains k_1, k_2, and k_3. Therefore, the procedure of implementing the controller is as following:

1. Acquire the current state of the vehicle from the input of the function
2. Convert the orientation of the vehicle from quaternion to euler angle
3. Compute the error vector
4. Compute the controller input (speed and steering angle velocity)
5. Pack the controller input to an AckermannDrive message and return the message

3.4 Combining the model and controller

After implementing the vehicle model and the waypoint following controller, the last step is to combine the two parts together. This should happen in the setModelState function. The procedure of implementing the function is as following:

1. Compute the controller input to the vehicle

2. Using the controller input from previous step to compute the rate of change for the state variables
3. Use the rate of change of the state variables to update state of the vehicle

After implementing the setModelState function, you can run the controller together with the vehicle model you implemented with the following procedure:

1. Start the Gazebo simulator
2. go to the ./src/mp2/src and run mp2.py with python 2

You should be able to test your implementation by sending target waypoint to the vehicle by using the command shown below. Note that the only thing you should change in the command is the target x, y position.

```
rostopic pub /gem/waypoint gazebo_msgs/ModelState "model_name: ''
pose:
  position:
    x: val
    y: val
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0
twist:
  linear:
    x: 0.25
    y: 0.25
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
reference_frame: ''"
```

3.5 Running with multiple waypoint and Plotting trajectory

In this part of the MP, you will track a series of given waypoints using the controller you implemented in last part. Note that it can be more difficult for the vehicle to track a series of waypoints because the orientation and speed when vehicle reaches a given waypoint can influence how the vehicle approach next waypoint. The provided waypoint list is coded in the constructor of the bicycleModel class. You can run multiple waypoints by uncommenting the corresponding part of the code and add the waypoints to the waypointList in class. You can also modify the waypoint list to run your own list of waypoint.

When the vehicle reaches the final waypoint, it will record the total amount of time it takes for your vehicle to traverse all waypoints. To get the amount of time it take to track provided waypoints, the vehicle should start from the default position when Gazebo is launched. Alternatively, you can reset the position of the vehicle by running the respawn.py file following command.

```
python respawn.py 0 0
```

Your controller should take less than 180s to reach all waypoints. You should play around with the gains to reduce the amount of time it takes to track all waypoints. Record the best time you get during the experiment.

In addition, you will need to plot the trajectory of the the vehicle is travelling. You should draw a x-y plot with the default initial position when you start gazebo simulator. In addition, you should mark the waypoints in your plot. The data points can be obtained by recording the x, y position of the vehicle using the getVehicleState function. There is no "the correct solution" for how to get the plot. Feel free to modify any code to achieve this.

4 Report

Each group should upload a short report (2 pages). First, all group members' names and netIds should be listed on the first page. Then, discuss the the following questions possibly with screenshots:

1. Briefly talk about how you choose the gain values for the waypoint following controller and also talk about how the change of gain value affect the amount of time it takes to track all given waypoints.
2. Some plots for the trajectory of the position of the vehicle tracking provided waypoints.
3. If anything is not working for this MP, do you have any thoughts on why it is not working?
4. Any feedback you have for SafeAutonomy498-team on the Lab and the MP?

5 Submission instructions

Please pack your report and your code in a zip file and submit to Compass 2G by one of the members in your group (homework need to be submitted individually). In the following week, each group should come to one of the office hours and demo the code to the TAs. The TA will primarily check if the vehicle can properly follow the waypoints. All members should show up and be prepared to answer some questions. The TA might pick a random person and ask that person to answer the question. So all students should be familiar with the content.

6 Grading rubric

- 15%: The vehicle model is working properly
- 25%: The vehicle can follow a single waypoint
- 25%: The vehicle can follow multiple waypoint
- 10%: TAs questions are answered properly during the demo.
- 25%: Report