EE3EY4: Electrical Systems Integration Project
Lab 6: Setting Up Simulation Environment and Manual Driving of McMaster AEV

Lab Section: L03
Instructor: Dr. Sirouspour

Aithihya Kompella - kompella - 400310794
Benji Richler - richlerb - 400296988
Shathurshika Chandrakumar - chands39 - 400315379
W. Tisuka Perera - pererw2 - 400318373

Date of Submission: March 8th, 2023

1) **Explain why each of these packages may be needed for implementation of the simulation environment.**

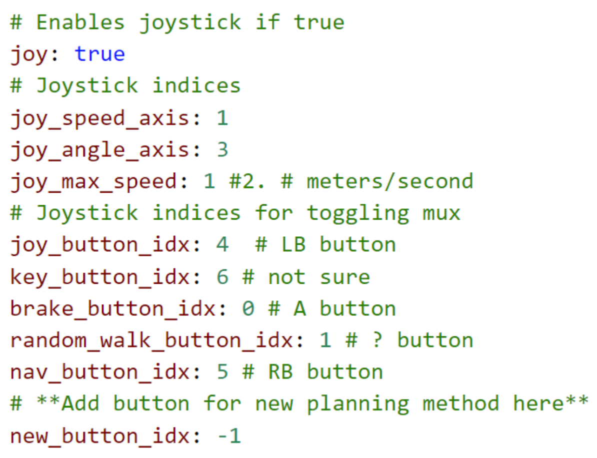
The *tf2_geometry_msgs* package allows the program to convert ROS messages to *tf2* messages, thus allowing it to extract data from those messages. Specifically *geometry_msgs*, which have data describing the position and orientation of objects in the ROS system are converted to *tf2* transform data. *tf2* data will be used in the simulation to manage and transform multiple coordinate frames over time. This package essentially helps the simulation convert between two vital data forms to manage and sense the surroundings of the AEV.

The *ackermann_msgs* package gives ROS messages for the vehicle using front-wheel Ackermann steering (a type of geometrical configuration for steering). The Ackernam steering is a system in which a single steering mechanism controls the front wheels. This package thus allows the ROS system to simulate the vehicle steering.

The *joy* package provides the *joy_node* which interfaces the joystick to ROS.T his will be used to interface the joystick commands of speed and steering in the simulation.

The *map_server* package gives access to map_server ROS node , which provides the ROS service of map data, and also allows dynamically generated maps to be saved to files. This will be used to generate and manage environment maps to create the simulation.

2) **By examining the content of the code provided to you, describe how the joystick action buttons are assigned.**



```
# Enables joystick if true
joy: true
# Joystick indices
joy_speed_axis: 1
joy_angle_axis: 3
joy_max_speed: 1 #2. # meters/second
# Joystick indices for toggling mux
joy_button_idx: 4  # LB button
key_button_idx: 6 # not sure
brake_button_idx: 0 # A button
random_walk_button_idx: 1 # ? button
nav_button_idx: 5 # RB button
# **Add button for new planning method here**
new_button_idx: -1
```

Figure 1: Joystick indice assignments from params.yaml

The joystick action buttons are used for manipulating the functions of the AEV. These functions are executed when the code receives the output list from the joystick, and checks what buttons have been pressed according to the output array sent by the AEV. It can be seen in Figure 1 that the array indices are assigned to specific buttons on the joystick. The joystick action button assignments is as follows:

LB - toggle joystick controller ON/OFF
RB - toggle navigation ON/OFF
Left Mini Stick - steering control
Right Mini Stick - speed control
A - toggle emergency break ON/OFF

B - toggle random walker ON/OFF
Back - toggle keyboard ON/OFF

3) **Why would you use a ROS Launch file?**

A ROS launch file is used when multiple nodes, and initialization parameters need to be set up. This is a quick and automated way of setting up the ROS environment, as opposed to manually typing each command.

4) **Briefly explain the role of each node launched by the above launch file.**

The nodes in the launch file above represent the necessary nodes for the racing simulation. The launch file's task is to start the multiple nodes that are essential for the F1TENTH simulator, all of which have specific roles to complete during the simulation. The nodes in question are as follows:

joy_node - listens to joystick messages, then converts them into ROS.

map_server - loads a map from the maps folder, providing it as a ROS service.

racecar_model.launch - launches a model of racecar in the simulator.

f1tenth_simulator - runs simulator with parameters specified in *params.yaml* file.

params.yaml - the kinematics and dynamics of the vehicle are modeled and accounts for the AEV motion commands and outputs based on the simulated sensor measurements.

mux_controller - a multiplexer that passes one of its many inputs to the */drive* topic based off the value of the selector signal in */mux*.

behavior_controller - creates control commands for the multiplexer based on the input results of the keyboard, joystick, or sensor observations.

random_walker - a standard controller used in the simulator that moves the vehicle randomly.

keyboard - reads inputs from the physical keyboard and is used for manually driving the car using the keyboard.

rviz - launches the RVIZ visualization tool along with the simulator.rviz file to visualize the simulation data in a 3D environment.

5) **Examine the computation graphs and compare them against the F1TENTH simulator software block diagram in Fig.1. Identify four nodes and four topics from the graph and briefly explain their role.**

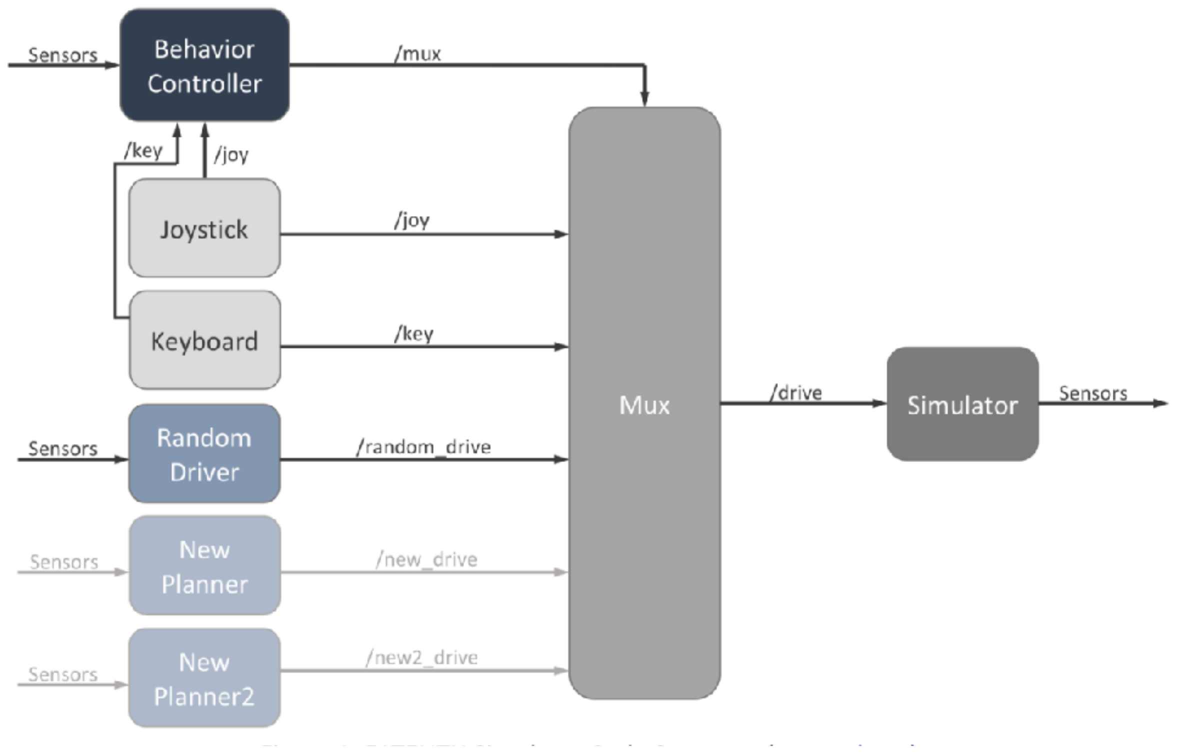


Figure:

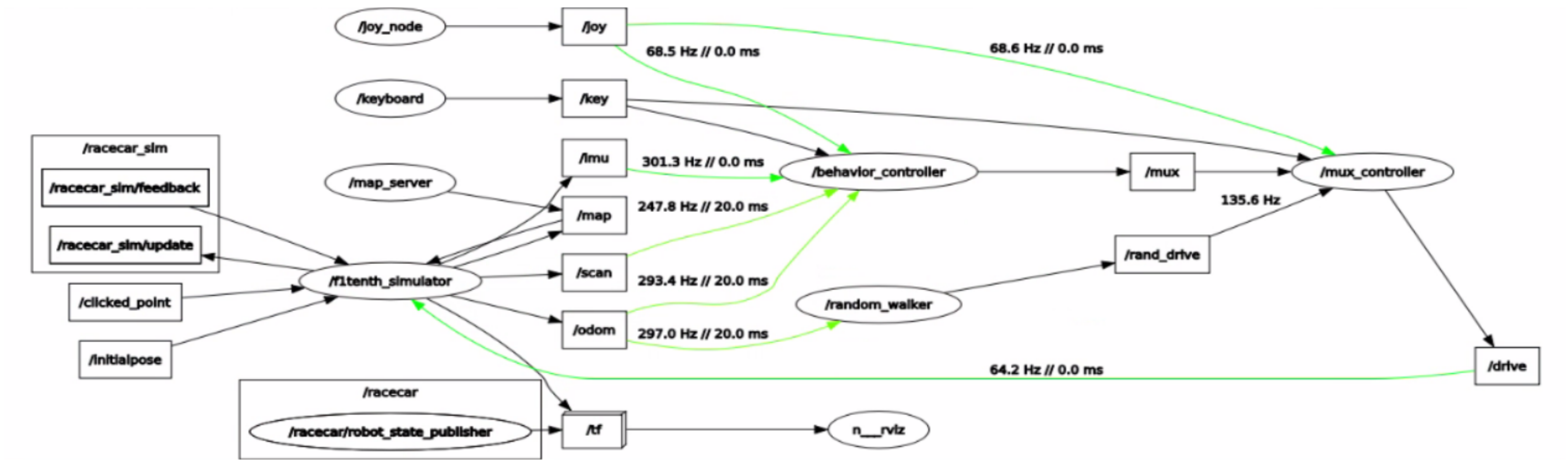


Figure:

The F1TENTH Simulator Code Structure has five nodes: Joystick, Keyboard, Random Driver, New Planner, and New Planner 2. These nodes publish messages to the Mux node, which then sends data to the simulator. Similarly, the computation graph has the nodes /joy_node, /keyboard, /random_walker nodes, which then publish messages to the /mux_controller. /mux_controller then sends the message to the /drive topic and then finally to /f1tenth_simulator. Essentially the Simulation Code Structure and the computation graph implement the same overall blocks, however the computation graph displays more detail of topics being published as well as additional topics such as /racecar_sim, and /racecar_sim/feedback.

Nodes:

/joy_node:

This node interfaces between the Linux Joystick and ROS. It publishes "Joy" messages to the /joy topic whose data controls the state of the joystick's buttons and axes.

/random_walker:
This node generates random driving commands, publishing them to the topic /rand_drive.

/behavior controller:
This node sends commands to the multiplexer based on keyboard (/key), joystick(/joy) or sensor(/scan) topic inputs

/mux_controller:
This node, subscribed to /mux topic, acts as a multiplexer by publishing a vehicle function command to /drive topic based on the input sent by joystick, keyboard, or random walk.

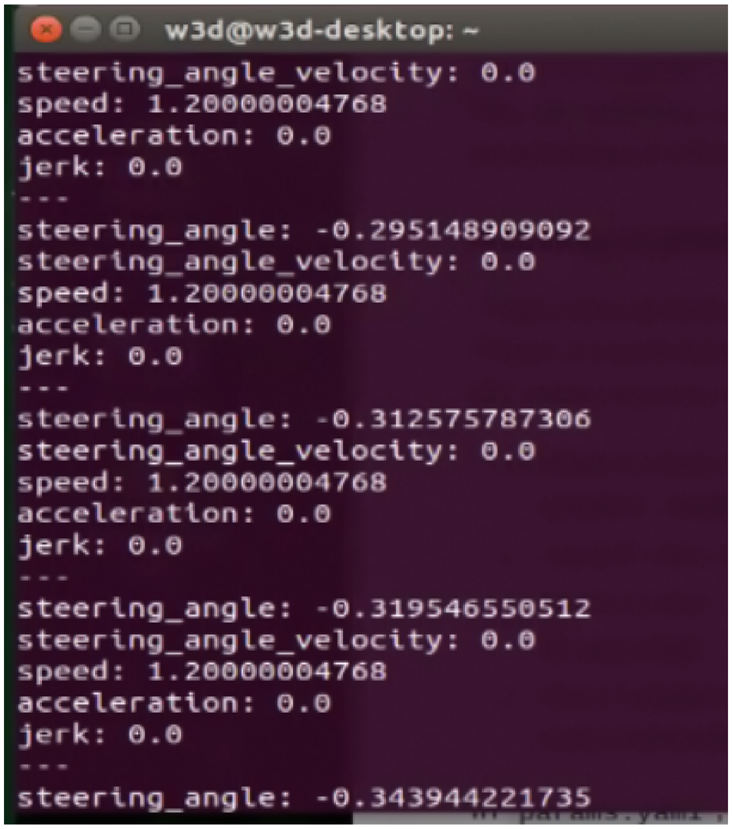
Topics:
/drive:
This topic passes the vehicle driving commands from the multiplexer to the simulator. Data such as steering angle and linear velocity are published in this topic.

/joy
This passes the state of the buttons and axes of the joystick controller in order to select the input signal - whether it be the keyboard, controller, or random walk, and to control the vehicle within the simulation via /mux_controller.

/mux
The node /behaviour_controller publishes to this topic, sending a selector signal to determine which input is passed into the /drive topic through the MUX. Essentially this topic feeds into a multiplexer that selects the input based on the selector signal fed into it.

/odom
This topic channels information about AEV's simulated odometry, carrying information about position, orientation, linear and angular velocities.

6) **What do you notice by examining the output of the drive topic? Which of the data fields are being published?**



```
steering_angle_velocity: 0.0
speed: 1.20000004768
acceleration: 0.0
jerk: 0.0
---
steering_angle: -0.295148909092
steering_angle_velocity: 0.0
speed: 1.20000004768
acceleration: 0.0
jerk: 0.0
---
steering_angle: -0.312575787306
steering_angle_velocity: 0.0
speed: 1.20000004768
acceleration: 0.0
jerk: 0.0
---
steering_angle: -0.319546550512
steering_angle_velocity: 0.0
speed: 1.20000004768
acceleration: 0.0
jerk: 0.0
---
steering_angle: -0.343944221735
```

The output of the drive topic shows the data for steering angle, steering angle velocity, speed, acceleration and jerk. This data is published by the */mux_controller* with the */f1tenth_simulator* is subscribed. It can be seen that other than steering angle and speed, the other parameters are zero. This means that only steering angle and speed are controlled within the simulation. This makes sense given the code provided in experiment.cpp, which has the publish_driver_command function that only controls the desired_speed and desired_steer_ang.

7) **The VESC driver can be set up to control the motor in speed, or torque/current modes. Comment on the differences between these two modes and their suitability for use in electrical vehicles in manual and self-driving modes.**

In speed mode, the VESC aims at maintaining a constant speed setpoint
- PID controller is used to adjust voltage applied to control motor speed
- The current drawn by the motor is proportional to the load. If the load is constant or the car is being driven manually
- The driver is responsible for maintaining a constant speed

In torque mode:
- Provides greater control over the motor
- q-axis current is manipulated to control motor's torque
- Higher speeds require less current to maintain a constant torque.
- The desired torque is the same regardless of the speed.
- As the motor speed increases, the current being drawn decreases.

Speed control mode is well-suited to manual driving mode as the driver requires easy control over the vehicle's speed through pressing the accelerator pedal. Simultaneously, VESC is able to automatically do torque adjustment to maintain the user's desired speed.

Torque mode is better suited to self-driving mode, as the vehicle's self-driving system requires more fine-tuned control over the motor's torque, which is needed for the vehicle to respond to different changes in driving environment.

8) **Examine the rate at which motor speed servo steering angle commands are published to their respective topics. What is the rate of execution for the VESC driver node "/vesc_driver_node"? What factors determine these rates?**

The average rate at which the motor speed servo steering angle commands are published to their respective topics is around 75 Hz. This is determined by the driver_smoother_rate parameter of 75.0 messages/sec set in the params.yaml file.

9) **Derive this formula:**

$k_\omega = \frac{15pg_r}{\pi r_\omega}$

Let v_a = actual linear speed of wheel

$$\frac{v_a}{v_s} = g_r$$

$$v_a = g_r v_s$$

Let ω_m = mechanical angular velocity of wheel (rad/s)

$$r_\omega \omega_m = v_a$$

$$r_\omega \omega_m = g_r v_s$$

$$\omega_m = \frac{g_r v_s}{r_\omega} \, rad/s$$

Converting rad/s to rpm (Multiply by 30/π)

$$\omega_m = \frac{30 g_r v_s}{\pi r_\omega} \, rpm$$

Convert mechanical angular velocity ω_m to electrical angular velocity ω_e , PP representing number of poles

$$\omega_e = PP\omega_m$$

$$PP = \frac{p}{2}$$

$$\omega_e = \frac{p}{2}\omega_m$$

$$\omega_e = \frac{p}{2} \times \frac{30 g_r v_s}{\pi r_\omega}$$

Therefore,

$$k_\omega = \frac{15pg_r}{\pi r_\omega}$$

10) **Let us assume that the actual gain is $k\delta = \alpha k\delta 0$, where $k\delta 0$ is the initial value given in "params.yaml" and α is an unknown adjustment factor. Find the value of α from the result of your calibration experiment and adjust the value of $k\delta$ accordingly.**

The offset value $\delta 0$ (speed_to_erpm_offset) was found to be 0.496 through trial and error to minimize the drifting of the AEV when driving on a straight path. The params.yaml showed that the max_steering_angle (k_δ δ) = 0.4189 rad, $k_{\delta 0}$ = -0.88, and wheelbase (l) = 0.287m. The AEV was driven in a circular motion, thus giving a diameter of 126.4cm or a radius of 63.2cm. The following calculations were used to find α.

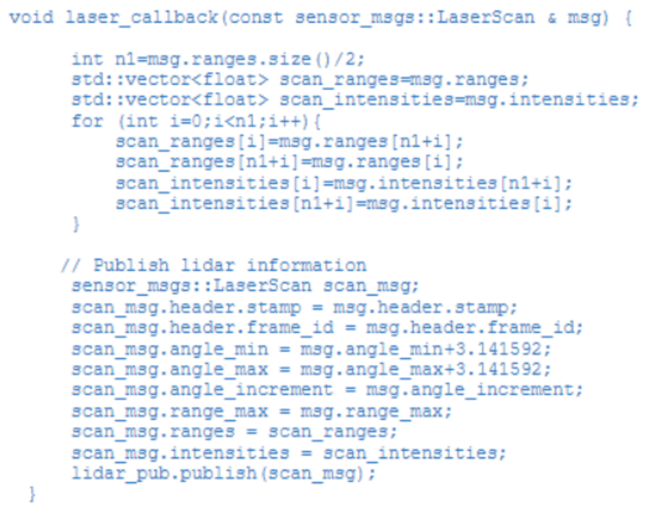
$$tan\delta \; = \; l/r \; = \; 0.287/0.632$$
$$\delta \; = \; 0.426$$
$$k_\delta = \alpha k_\delta \delta$$
$$\alpha = -0.4189/\delta k\delta$$
$$\alpha = -0.4189/(0.426*-0.88)$$
$$\alpha \; = \mathbf{1.117}$$

11) Explain what this function does.

```
 void publish_driver_command( const ros::TimerEvent&){

        double desired_delta = desired_steer_ang-last_servo;
        double clipped_delta = std::max(std::min(desired_delta,
max_delta_servo), -max_delta_servo);
        double smoothed_servo = last_servo + clipped_delta;
        last_servo = smoothed_servo;
        servo_msg.data=smoothed_servo;


        double desired_rpm = desired_speed-last_rpm;
        double clipped_rpm = std::max(std::min(desired_rpm, max_delta_rpm), -
max_delta_rpm);
        double smoothed_rpm = last_rpm + clipped_rpm;
        last_rpm = smoothed_rpm;
        erpm_msg.data=smoothed_rpm;

        servo_pub.publish(servo_msg);
        erpm_pub.publish(erpm_msg);
    }
```

This function calculates and smoothes out the values for the desired angle and RPM and publishes to their necessary topics. The delta values are calculated by getting the difference between the desired value and the last value, and are then clipped to ensure that in the event of sudden large changes in the commands, the system will be stable and not be damaged.

12) Explain what this function achieves.



```
 void laser_callback(const sensor_msgs::LaserScan & msg) {

      int n1=msg.ranges.size()/2;
      std::vector<float> scan_ranges=msg.ranges;
      std::vector<float> scan_intensities=msg.intensities;
      for (int i=0;i<n1;i++){
          scan_ranges[i]=msg.ranges[n1+i];
          scan_ranges[n1+i]=msg.ranges[i];
          scan_intensities[i]=msg.intensities[n1+i];
          scan_intensities[n1+i]=msg.intensities[i];
      }

     // Publish lidar information
      sensor_msgs::LaserScan scan_msg;
      scan_msg.header.stamp = msg.header.stamp;
      scan_msg.header.frame_id = msg.header.frame_id;
      scan_msg.angle_min = msg.angle_min+3.141592;
      scan_msg.angle_max = msg.angle_max+3.141592;
      scan_msg.angle_increment = msg.angle_increment;
      scan_msg.range_max = msg.range_max;
      scan_msg.ranges = scan_ranges;
      scan_msg.intensities = scan_intensities;
      lidar_pub.publish(scan_msg);
  }
```

When executed, this function above is called constantly as it takes the most recent speed and steering angle commands, its values being stored in *desired_speed* and *desired_steer_ang*, respectively. It then publishes them to the topics */commands/motor/speed* and */commands/servo/position* for the VESC driver. These values are then "clipped" which happens to make sure the fluctuation in the speed and servo position doesn't exceed a threshold everytime the function is called. However if it does exceed the threshold, the change in motor speed and/or servo position is set to the maximum allowable range of change, whether it be positive or negative.

13) Explain what this part of the code does. Execute the following command:

```
update_command =
n.createTimer(ros::Duration(1.0/driver_smoother_rate),
&RacecarExperiment::publish_driver_command, this);
```

This part of the code creates a timer object which is used to ensure that the commands being sent to the driver system are at a consistent rate and maintaining smooth operation. The publish_driver_command function of the RacecarExperiment class is called at the rate of driver_smoother_rate. This is what the update_command object is repeatedly doing.

-END-