WildcatRacing

Release 1.0.0

Trevin Lake, Ryan Reidhead

Dec 13, 2022

CONTENTS

Welcome to Wildcat Racing! Here you will find documentation for Weber State University's F1Tenth 2022 Race Team.

CHAPTER

ONE

ARCHITECTURE

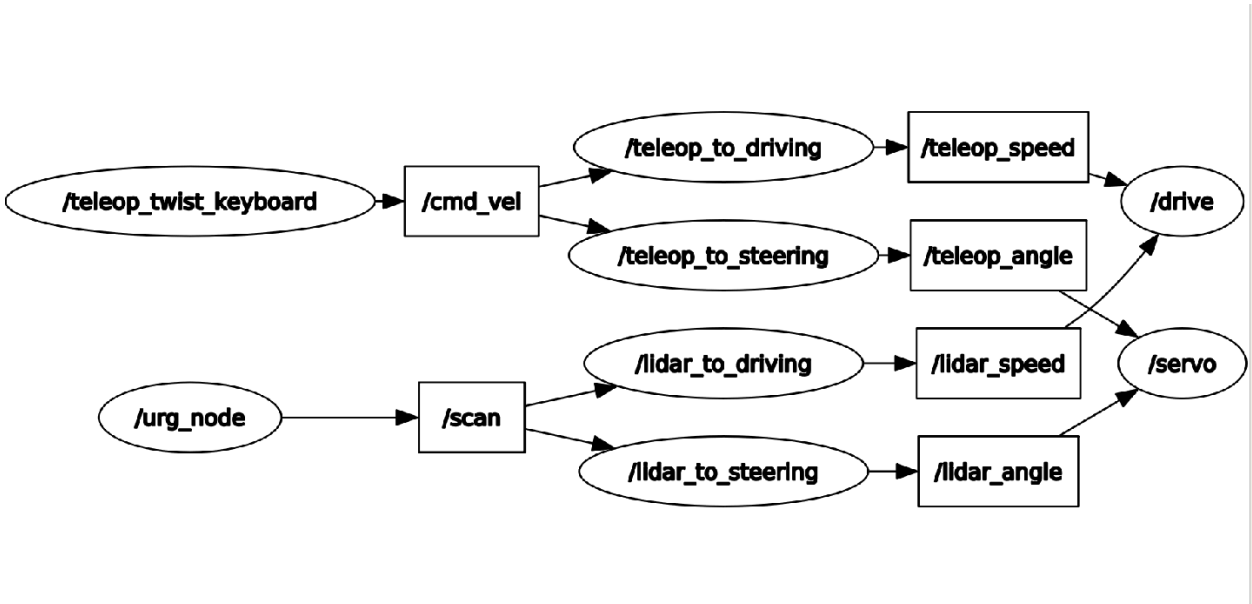
1.1 SysML Diagrams

CHAPTER

TWO

ROS CODE

2.1 RQT Graph



The above image is a graph of all ROS Nodes used within our vehicle and how they relate to each other. This RQT Graph should be referenced as the code below is examined. The ovals depict a ROS node and the rectangles depict a ROS topic.

2.2 ROS Node Data Characterization Example - Lidar to Steering Subscriber/Publisher Node

```
#! /usr/bin/env python3.7
# -*- coding: utf-8 -*-
import rospy
from std_msgs.msg import Empty
from std_msgs.msg import Float32
from sensor_msgs.msg import LaserScan
import numpy as np

```

(continues on next page)

(continued from previous page)

```
class LIDARAngle:

    # LIDAR info:
    # The LIDAR can 'see' as close as 0.2 meters (about 0.65ft)
    # and as far away as 30 meters (about 98.5ft).

    def __init__(self):
        self.angle = 0
        self.pub = rospy.Publisher('/lidar_angle', Float32, queue_size=35)
        self.scan_subscriber = rospy.Subscriber('/scan', LaserScan, self.callback)

    def callback(self, LaserScan):
        rospy.loginfo(rospy.get_caller_id() + "Latest LiDAR scan time was: %s\n",␣
↪LaserScan.scan_time)
        new_angle = Float32()
        ranges_arr = np.array(LaserScan.ranges)
        angle_count = 8
        right_deg_arr = np.zeros(600)
        left_deg_arr = np.zeros(600)
        mid_deg_arr = np.zeros(600)
        #angle_count is how many ranges per degree of angle change.
        #average ranges per angle_count both for 'smoothing' sensor data and
        #for simplifying ranges indeces to coincide with integer degrees.
        deg_avg = 0
        deg_sum = 0
        for i in range(360,960): #Look at right side of LiDAR
            right_deg_arr[i-360] = ranges_arr[i]
        for i in range(1200,1800): #Look at left side of LiDAR
            left_deg_arr[i-1200] = ranges_arr[i]
        for i in range(960,1200): #Look straight ahead of LiDAR
            mid_deg_arr[i-960] = ranges_arr[i]

        #The index of the min range will be 90 degrees away from where we want
        #to steer the car
        try: #for only one minimum index
            right_min_range_index = np.argmin(right_deg_arr)
            left_min_range_index = np.argmin(left_deg_arr)
            mid_min_range_index = np.argmin(mid_deg_arr)
            auto_brake_steer_arr = np.extract(mid_deg_arr < 1, mid_deg_arr)

            #if right and left sides are within .1m (4 inches) default to
            #90 degrees
            right_left_range_diff = right_deg_arr[right_min_range_index] - left_deg_
↪arr[left_min_range_index]
            #print("right_min_range_index: " + str(right_min_range_index))
            #print("left_min_range_index: " + str(left_min_range_index))
            #print("Difference: " + str(right_left_range_diff))
            #print("mid_min_range: " + str(mid_deg_arr[mid_min_range_index]))
            if np.abs(right_left_range_diff) < .1:
                new_angle = 90
            elif right_left_range_diff < 0: #turn left
                new_angle = (int((right_min_range_index + 360 + 720)/8)-45) #output(90,
```

(continues on next page)

(continued from previous page)

```
↪165)
                new_angle = ((new_angle - 90)*1.5) + 90
                print(str(right_min_range_index))
            elif right_left_range_diff > 0: #turn right
                print(str(left_min_range_index))
                new_angle = (int(((left_min_range_index + 1200 + 720)/8)-225))
↪#output(15,90)
                new_angle = ((new_angle - 90)*1.5) + 90
            else:
                new_angle = 90

            if new_angle > 180:
                new_angle = 180
            if new_angle < 0:
                new_angle = 0

        except: #for multiple minimum indeces, choose first minimum index
            right_min_range_index = np.argmin(right_deg_arr)[0]
            left_min_range_index = np.argmin(left_deg_arr)[0]
            #print("right_min_range_index: " + str(right_min_range_index))
            #print("left_min_range_index: " + str(left_min_range_index))
            #print("mid_min_range: " + str(mid_deg_arr[mid_min_range_index]))
            #if right and left sides are within .1m (4 inches) default to
            #90 degrees
            right_left_range_diff = right_deg_arr[right_min_range_index] - left_deg_
↪arr[left_min_range_index]
            #print("Difference: " + str(right_left_range_diff))
            if np.abs(right_left_range_diff) < .1:
                new_angle = 90
            elif right_left_range_diff < 0: #turn left
                new_angle = (int((right_min_range_index + 360 + 720)/8)-45) #output(90,
↪165)
                new_angle = ((new_angle - 90)*1.5) + 90
                print(str(right_min_range_index))
            elif right_left_range_diff > 0: #turn right
                print(str(left_min_range_index))
                new_angle = (int(((left_min_range_index + 1200 + 720)/8)-225))
↪#output(15,90)
                new_angle = ((new_angle - 90)*1.5) + 90
            else:
                new_angle = 90

            if new_angle > 180:
                new_angle = 180
            if new_angle < 0:
                new_angle = 0

        self.angle = new_angle
        self.pub.publish(new_angle)

if __name__ == '__main__':
    print("Running lidar_to_steering node.")
```

(continues on next page)

(continued from previous page)

```
106     rospy.init_node('lidar_to_steering')
107     rate = rospy.Rate(30) # 30Hz, max for our LIDAR is 40Hz
108     LIDARAngle()
109     rospy.spin()
```

Line 1 is the "shebang" which tells the Python interpreter which version of Python should be used to run the code. In line 17, we set this node to publish on a topic called lidar_angle. (In line 107, we can see it will be published at a rate of 30 Hz). Line 18 is where the node is set to subscribe to the /scan topic where our lidar data is published. In lines 33-38, we extract several subarrays from the total ranges_array provided from the ranges found in the /scan topic. One sub-array coincides with a snippet of ranges found on the right-hand side of the vehicle, another is a snippet of ranges found on the vehicle's left-hand side. Finally, the last snippet of ranges corresponds with ranges found directly ahead of the vehicle. Line 55 shows that if the minimum distance of both right and left sub-arrays differ by less than .1 meter, the vehicle defaults to 90 degrees which corresponds with the vehicle heading straight. Line 57 shows how the vehicle will make a left turn. Note: The vehicle uses the equation found in line 58 to determine the angle of the nearest obstacle in reference to the vehicle. Line 59 shows how the vehicle then offsets its heading by adjusting the steering 90 degrees away from the angle solved for in line 58. Lines 68-71 accounts for adjustments to the steering data characterization equations in how aggressive the steering response should be. For our vehicle, the steering response is static, however with dynamic steering response, it is possible to get invalid values returned from the characterization equations. These lines allow for a dynamic steering characterization while sanitizing the angles returned prior to publishing them to the lidar_angle topic. Line 73 is the except case for when there are multiple minimum indeces returned from lines 43-45. We chose to use the first minimum angle for this case. In line 102, we publish the characterized angle to the lidar_angle topic.

2.3 ROS Node Control Example - Steering Servo Subscriber Node

```
1  #! /usr/bin/env python3.7
2  # -*- coding: utf-8 -*-
3  import rospy
4  from adafruit_servokit import ServoKit
5  from std_msgs.msg import Float64
6  from std_msgs.msg import Float32
7  import numpy as np
8
9  kit = ServoKit(channels=16)
10 kit.servo[0].set_pulse_width_range(1400, 1825)
11
12 class Servo:
13
14     # Steering Servo info:
15     # The steering servo as configured above will take range 0-180
16     # where 0 is RHS and 180 is LHS.
17
18     def __init__(self):
19         self.angle = 0
20         self.teleop_angle_subscriber = rospy.Subscriber('/teleop_angle', Float64, self.
↪teleop_callback)
21         self.lidar_angle_subscriber = rospy.Subscriber('/lidar_angle', Float32, self.
↪lidar_callback)
22
23     def teleop_callback(self, msg):
```

(continues on next page)

(continued from previous page)

```
24          rospy.loginfo(rospy.get_caller_id() + "Latest teleop_angle was: %s\n", msg)
25          new_angle = Float64()
26          if msg != 0:
27              new_angle = round(msg.data)
28              self.angle = msg
29              kit.servo[0].angle = new_angle
30          else:
31              self.angle = 0
32
33      def lidar_callback(self, msg):
34          rospy.loginfo(rospy.get_caller_id() + "Latest lidar_angle was: %s\n", msg)
35          new_angle = Float32()
36          #Priority is given to teleop since self.angle is only updated by teleop.
37          if self.angle == 0:
38              new_angle = msg.data
39              kit.servo[0].angle = new_angle
40
41  if __name__ == '__main__':
42      print("Running servo node.")
43      rospy.init_node('servo')
44      Servo()
45      rospy.spin()
```

In Lines 9-10, we use the adafruit_servokit library to create a servo controller object that has 16 channels. This how the PCA9685 Servo Driver board is integrated in software. The pulse width chosen gives our car the full steering range of motion without locking up the servo. The values show in line 10 were found via trial and error. Line 12 is where we create a Servo class. We decided to make all Python ROS Nodes using object oriented architecture since Python's implementation of ROS in scripted architecture requires some 'tricks' and global variables to access and pass the variables even within the same Node. Lines 20-21 are where we tell this node to subscribe to both /teleop_angle and /lidar_angle topics. Line 23 is the teleop callback function which is called any time new data shows up on the /teleop_angle topic. In line 28, we can see that if a teleop message is not 0, we assign the nodes 'angle' attribute to be the value of the message. Then in line 29, we send the new angle to the steering servo over the PCA9685 control board at channel 0 using PWM via I2C protocol. Line 33 is the lidar callback function which is called every time new data appears on the /lidar_angle topic. Line 37 shows how we give priority to teleop since we are using teleop as an emergency override to stop autonomous driving using keyboard controls. Line 39 sends the latest lidar data to the steering servo via the PCA6985 servo control board.

Note: The teleop_callback is expected to be called at 120 Hz as the publisher node which publishes to the teleop_angle topic refreshes at 120 Hz. The lidar_callback is expected to be called at 30 Hz according to the lidar_angle topic's publish rate.

CHAPTER

THREE

LAUNCHER GUI

This launcher script runs ROS commands on the backend using Python's OS library. It launches the launch files found in the 'launch' folder. When the User presses Ctrl+C to kill the currently running launch mode, the script returns to the main menu after closing applicable bash windows. If the User exits the laucher GUI by entering '4' the GUI will close and return to a normal bash. The launcher script must be ran in a shell with root access (using the command sudo -s after opening a bash). This is because some of the scripts ran in the launch files need root access. For example, the LIDAR Ethernet configuration has to be ran as root in order to configure the ethernet port.

CHAPTER

FOUR

FIX-IT SCRIPTS

4.1 "Blinky" drive_motor_endpoint.py

This refers to a state that the Electronic Speed Controller gets stuck in a mode where it waits for calibration input from a remote controller. This is due to it being traditionally used in an RC setting. Since our vehicle does not utilize that funcionality, we wrote a script to send data directly to the ESC to knock it out of this calibration mode before launching our race modes.

```
#! /usr/bin/env python3.7
# -*- coding: utf-8 -*-
from adafruit_servokit import ServoKit
import numpy as np
import time

kit = ServoKit(channels=16)
command = ""
while command != "end":
    command = input("enter speed:")
    try:
      kit.continuous_servo[2].throttle = float(command)
    except:
        command = "end"

kit.continuous_servo[2].throttle = 0
```

4.2 servo_and_drive_calbration.py

This code was used to initially interface with the servo and drive motor directly rather than through a ROS node. It was used to iterate through and identify values which we could use to properly set the steering servo PWM pulse width to make the full range 0-180 degrees match the full range of steering where 0 degrees is a full right turn and 180 degrees is a full left turn and 90 degrees is straight ahead. The drive motor is treated as a continuous servo with values ranging from -1,1 where 0 is full coast, -1 is full brake, and 1 is full throttle.

```
#! /usr/bin/env python3.7
# -*- coding: utf-8 -*-
from adafruit_servokit import ServoKit
import numpy as np
import time

```

(continues on next page)

(continued from previous page)

```
kit = ServoKit(channels=16)

kit.servo[0].set_pulse_width_range(1400, 1825)

"""
#Use this code to set pulse width min and max.
kit.servo[0].set_pulse_width_range(1000, 2000)
"""

"""
#Use this code to set drive motor throttle; 0 is stopped,
0.5 is half speed, 1 is full speed.
kit.continuous_servo[2].throttle = 0
"""

"""
#Use this code to set the steering servo's angle.
kit.servo[0].angle = 180
"""

"""
#Use this code to set the steering servo's actuation range.
kit.servo[0].actuation_range = 180
"""

#Steering Constants
STEERING_MIN_ANGLE = 0
STEERING_MAX_ANGLE = 180
STEERING_STEP = 0.2
STEERING_STEP_LIST = np.arange(STEERING_MIN_ANGLE, STEERING_MAX_ANGLE, STEERING_STEP)
INVERSE_STEERING_STEP_LIST = np.arange(STEERING_MAX_ANGLE, STEERING_MIN_ANGLE, STEERING_STEP)

#Drive Constants
DRIVE_THROTTLE_MIN = 0
DRIVE_THROTTLE_MAX = 0.05
DRIVE_STEP = 0.001
DRIVE_STEP_LIST = np.arange(DRIVE_THROTTLE_MIN, DRIVE_THROTTLE_MAX, DRIVE_STEP)


#Sweep the steering servo within its entire range.
for step in STEERING_STEP_LIST:
    kit.servo[0].angle = step
    print(step)
    time.sleep(.01)

time.sleep(5)

#Slowly increase throttle on drive motor within its entire range.
for step in DRIVE_STEP_LIST:
    kit.continuous_servo[2].throttle = step
    time.sleep(.1)
```

(continues on next page)

(continued from previous page)

```
    print(step)



kit.continuous_servo[2].throttle = 0

kit.servo[0].angle = 90
```

4.3 real_time_hall_effect.py

This code is the real-time graphing of the suspension sensor data using matplotlib's animate function.

```
#!/usr/bin/env python3.7
import matplotlib.pyplot as plt
import numpy as np
import time
import adafruit_ads1x15.ads1115 as ADS
import board
import busio
import os
i2c = busio.I2C(board.SCL_1, board.SDA_1)
import sys
sys.path.append('../')
import time
from adafruit_ads1x15.analog_in import AnalogIn
import matplotlib.animation as animation
from matplotlib.ticker import FuncFormatter
import datetime as dt

def init():
    line.set_ydata([np.nan] * len(x))
    return line,

# This function is called periodically from FuncAnimation
def animate(i, xs0, ys0, xs1, ys1, xs2, ys2, xs3, ys3):

    # Read voltage from i2c_bus
    voltage_0 = hall_0.voltage
    voltage_1 = hall_1.voltage
    voltage_2 = hall_2.voltage
    voltage_3 = hall_3.voltage

    # Add x and y to lists
    xs0.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
    ys0.append(voltage_0)

    xs1.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
    ys1.append(voltage_1)

    xs2.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
```

(continues on next page)

(continued from previous page)

```
        ys2.append(voltage_2)

        xs3.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
        ys3.append(voltage_3)

        # Limit x and y lists to 20 items
        xs0 = xs0[-20:]
        ys0 = ys0[-20:]

        xs1 = xs1[-20:]
        ys1 = ys1[-20:]

        xs2 = xs2[-20:]
        ys2 = ys2[-20:]

        xs3 = xs3[-20:]
        ys3 = ys3[-20:]

        # Draw x and y lists
        ax[0][0].clear()
        ax[0][0].plot(xs3, ys3, 'g')
        ax[0][0].set_title("Front Left Suspension Sensor")
        ax[0][0].tick_params(labelrotation=45)


        ax[0][1].clear()
        ax[0][1].plot(xs2, ys2, 'k')
        ax[0][1].set_title("Front Right Suspension Sensor")
        ax[0][1].tick_params(labelrotation=45)

        ax[1][0].clear()
        ax[1][0].plot(xs1, ys1, 'orange')
        ax[1][0].set_title("Rear Left Suspension Sensor")
        ax[1][0].tick_params(labelrotation=45)

        ax[1][1].clear()
        ax[1][1].plot(xs0, ys0, 'm')
        ax[1][1].set_title("Rear Right Suspension Sensor")
        ax[1][1].tick_params(labelrotation=45)


        # Format plot
        plt.subplots_adjust(bottom=0.30)

        for a in ax.flat:
            a.set(xlabel='Time', ylabel='Voltage')
        plt.tight_layout()


    sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
    ADS1115_REG_CONFIG_PGA_6_144V        = 0x00 # 6.144V range = Gain 2/3
    ADS1115_REG_CONFIG_PGA_4_096V        = 0x02 # 4.096V range = Gain 1
```

(continues on next page)

(continued from previous page)

```
ADS1115_REG_CONFIG_PGA_2_048V        = 0x04 # 2.048V range = Gain 2 (default)
ADS1115_REG_CONFIG_PGA_1_024V        = 0x06 # 1.024V range = Gain 4
ADS1115_REG_CONFIG_PGA_0_512V        = 0x08 # 0.512V range = Gain 8
ADS1115_REG_CONFIG_PGA_0_256V        = 0x0A # 0.256V range = Gain 16
ads1115 = ADS.ADS1115(i2c)

hall_0 = AnalogIn(ads1115, ADS.P0)
hall_1 = AnalogIn(ads1115, ADS.P1)
hall_2 = AnalogIn(ads1115, ADS.P2)
hall_3 = AnalogIn(ads1115, ADS.P3)

# set x axis for time
hall_0_list = []
hall_1_list = []
hall_2_list = []
hall_3_list = []
time0_list = []
time1_list = []
time2_list = []
time3_list = []
# Create figure for plotting
fig, ax = plt.subplots(2,2)
xs0 = []
ys0 = []
xs1 = []
ys1 = []
xs2 = []
ys2 = []
xs3 = []
ys3 = []

while True:
    #Get the Digital Value of Analog of selected channel
    #print(hall0.value, hall0.voltage)
    hall_0_list.append(hall_0.voltage)
    time0_list.append(time.time())
    time.sleep(0.02)
    hall_1_list.append(hall_1.voltage)
    time1_list.append(time.time())
    time.sleep(0.02)
    hall_2_list.append(hall_2.voltage)
    time2_list.append(time.time())
    time.sleep(0.02)
    hall_3_list.append(hall_3.voltage)
    time3_list.append(time.time())
    #print("A0:%dmV A1:%dmV A2:%dmV A3:%dmV"%(adc0_list[i],adc1_list[i],adc2_list[i],
↪adc3_list[i]))




    # Set up plot to call animate() function periodically
```

(continues on next page)

(continued from previous page)

```
    ani = animation.FuncAnimation(fig, animate, fargs=(xs0, ys0, xs1, ys1, xs2, ys2, xs3, ys3), interval=100)
    #ani_1 = animation.FuncAnimation(fig, animate, fargs=(xs1, ys1), interval=1000)
    #ani_2 = animation.FuncAnimation(fig, animate, fargs=(xs2, ys2), interval=1000)
    #ani_3 = animation.FuncAnimation(fig, animate, fargs=(xs3, ys3), interval=1000)
    plt.tight_layout()
    plt.xticks(rotation=45, ha='right')
    plt.show()
```

4.4 real_time_IMU.py

This code is the self-written driver code for accessing data stored on the ADIS16470 IMU's registers. Note: The baud rate is likely incorrect as we could not readily find the proper one to use.

```
#!/usr/bin/env python3.7
import board
import busio
import digitalio
import array
import numpy as np
from adafruit_bus_device.spi_device import SPIDevice
import struct
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.ticker import FuncFormatter
import datetime as dt
import time

# This function is called periodically from FuncAnimation
def animate(i, xs, ys):

    # Add x and y to lists
    xs.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
    ys.append(x_accel)

    # Limit x and y lists to 20 items
    xs = xs[-20:]
    ys = ys[-20:]

    # Draw x and y lists
    ax.clear()
    ax.plot(xs, ys)

    # Format plot
    plt.xticks(rotation=45, ha='right')
    plt.subplots_adjust(bottom=0.30)
    plt.title('Acceleration Over Time')
    plt.ylabel("m/(s^2)")
    return

```

(continues on next page)

(continued from previous page)

```

#Constants
X_GYRO_LOW = bytearray([0x04,0x05])
X_GYRO_OUT = bytearray([0x06, 0x07])
Y_GYRO_LOW = bytearray([0x08,0x09])
Y_GYRO_OUT = bytearray([0x0A, 0x0B])
Z_GYRO_LOW = bytearray([0x0C,0x0D])
Z_GYRO_OUT = bytearray([0x0E, 0x0F])
X_ACCEL_LOW = bytearray([0x10,0x11])
X_ACCEL_OUT = bytearray([0x12, 0x13])
Y_ACCEL_LOW = bytearray([0x14,0x15])
Y_ACCEL_OUT = bytearray([0x16, 0x17])
Z_ACCEL_LOW = bytearray([0x18,0x19])
Z_ACCEL_OUT = bytearray([0x1A, 0x1B])
TEMP_OUT = bytearray([0x1C, 0x1D])
TIME_STAMP = bytearray([0x1E, 0x1F])


#Setup spi bus
spi = busio.SPI(board.SCLK, MISO=board.MISO, MOSI = board.MOSI)
#Setup Chip Select
cs = digitalio.DigitalInOut(board.CE0_1)

#Create an instance of the SPIDevice class
device = SPIDevice(spi, cs, baudrate=4000, polarity=0, phase=0)


def spi_request_float32(request_low, request_out):
    result_low = bytearray(2)
    result_out = bytearray(2)
    result = bytearray(4)
    with device as spi:
        spi.write_readinto(request_low, result_low)
        spi.write_readinto(request_out, result_out)
    result_out.extend(result_low)
    result_float = struct.unpack('f', result_out)

    return result_float[0]

def spi_request_decimal(request):
    result = bytearray(2)
    with device as spi:
        spi.write_readinto(request, result)

    result_decimal = int.from_bytes(result, byteorder='big', signed=True)

    return result_decimal
time_stamp = 0
# Create figure for plotting
fig = plt.figure()
ax = fig.add_subplot(2, 2, 1)
xs = []
```

(continues on next page)

(continued from previous page)

```
ys = []
x_accel_list = []
x_gyro_list = []
time0_list = []
while True:
    x_accel = spi_request_float32(X_ACCEL_LOW, X_ACCEL_OUT)
    y_accel = spi_request_float32(Y_ACCEL_LOW, Y_ACCEL_OUT)
    z_accel = spi_request_float32(Z_ACCEL_LOW, Z_ACCEL_OUT)

    x_gyro = spi_request_float32(X_GYRO_LOW, X_GYRO_OUT)
    y_gyro = spi_request_float32(Y_GYRO_LOW, Y_GYRO_OUT)
    z_gyro = spi_request_float32(Z_GYRO_LOW, Z_GYRO_OUT)

    temp = spi_request_decimal(TEMP_OUT)/10
    time_stamp += spi_request_decimal(TIME_STAMP)
    print("x_acceleration: " + str(x_accel))
    print("x_gyro: " + str(x_gyro))
    print("Temperature: " + str(temp) + "\n")
```

4.5 lidar_ethernet_setup.py

This code configure's the Jetson Nano's Ethernet port for the Lidar data rather than being used for network information.

```
#! /usr/bin/env python3.7
# -*- coding: utf-8 -*-

import os
import rospy

os.system("sudo ip addr add 192.168.0.15/24 broadcast 192.168.0.255 dev eth0")
os.system("rosrun urg_node urg_node _ip_address:=192.168.0.10")

print("LIDAR Ethernet port has been configured")
```

CHAPTER

FIVE

LIDAR

CHAPTER

SIX

INTERTIAL MEASUREMENT UNIT

```
x_acceleration: 0.0
x_gyro: 2.5290782398958475e-29
Temperature: 12.7
```

The IMU was integrated using self-written driver code. The baud rate likely needs to be updated to a more correct value since "valid" entries are only present every few loops. There is a ton of potential to improve upon the utilization of the IMU's data. Once characterized, it can be used to bridge the gap of data 'blindness' in between the LIDAR data cycles. The IMU can also serve to help with localization when coupled with the car's other systems.

CHAPTER

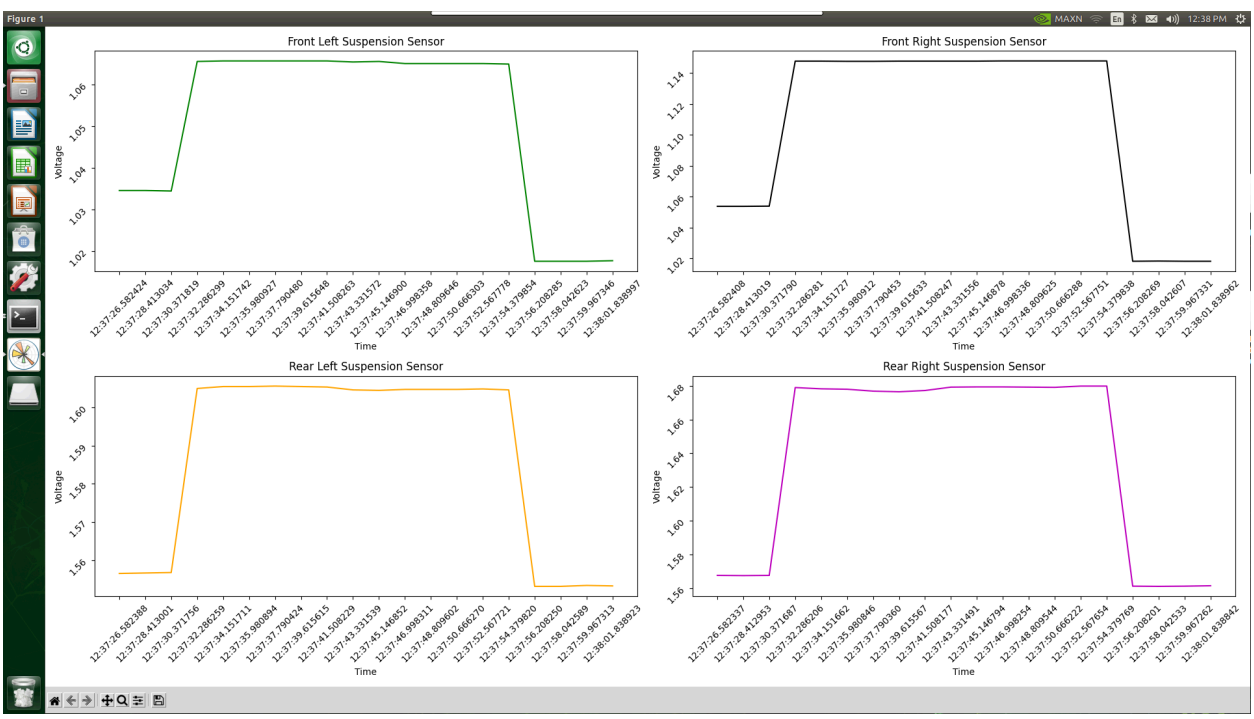
SEVEN

HALL EFFECT

7.1 Integration

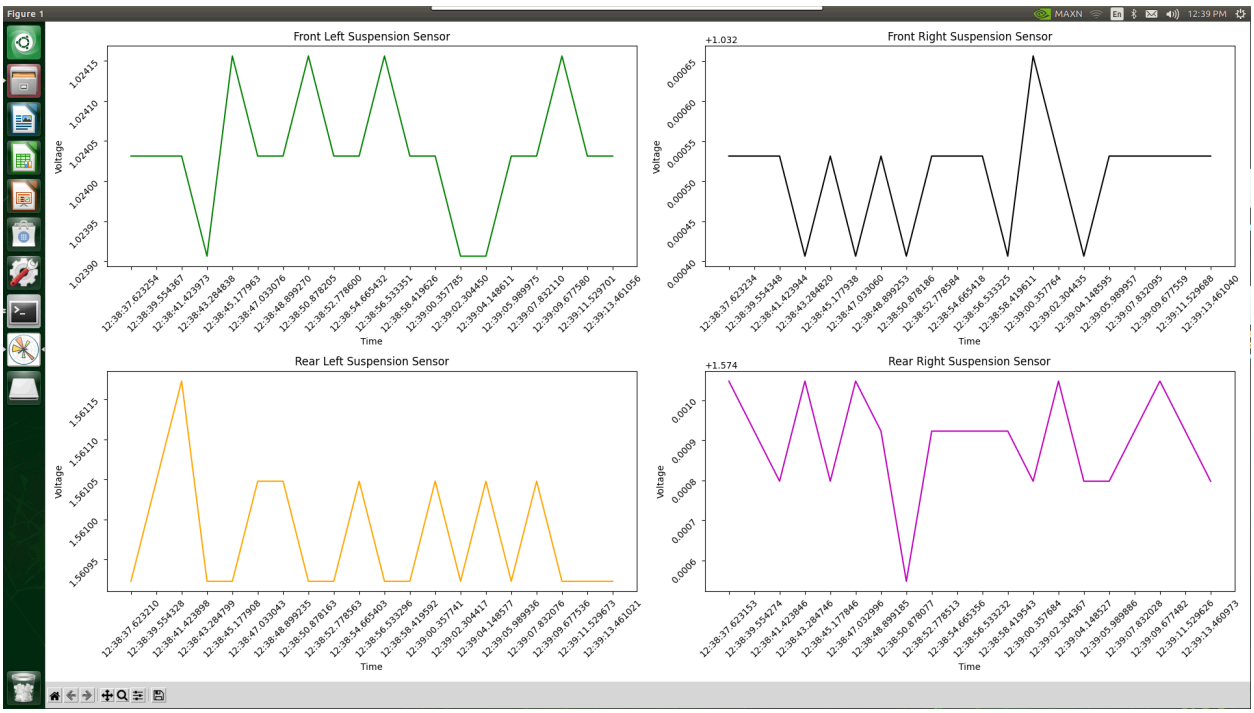
The Hall Effect sensors were integrated using an ADS1115 16 bit ADC. The adafruit_ads1x15.ads1115 library was utilized to read the voltages on each channel and then the outputs for each channel are graphed using a real-time plot generated using matplotlib's animate function. The plots below show raw voltages from the sensors.

7.2 Full Range of Suspension Travel



The full range of motion can be seen on the above graph when the vehicle's suspension is fully compressed on all 4 wheels and then the vehicle is lifted off of the ground to show the suspensions full range. Here we see an average range of about 120mV with each sensor which corresponds with the vehicle's maximum and minimum ride heights.

7.3 Noise



In this image, we see that each Hall Effect sensor's noise floor. This average is about 200 nV. This is great because it means valid suspension travel data will be easily detectable and discernable from the sensor's noise floor, even without amplifying the signal.

CHAPTER

EIGHT

BILL OF MATERIALS

CHAPTER

NINE

SCHEMATICS

CHAPTER

TEN

DATASHEETS

CHAPTER

ELEVEN

3D PRINTED PARTS