1. **Components Used:**
    - Nvidia Jetson Nano 2GB
    - Teensy 4.0
    - Large Pololu Frame
    - 65mm TT Motor Wheels x4
    - TT Motor with Encoder x4
    - Feather Board Motor Driver
    - Battery 18650 x 4
    - BNO055 IMU

*see video link below for demo of all tasks

2. **Task 1 - Tick Publisher through ROS on Jetson Nano via Teensy**

```
#include <ros.h>
#include <std_msgs/Int16.h>
#include <geometry_msgs/Twist.h>
#include <Wire.h>
#include <Adafruit_MotorShield.h>

Adafruit_MotorShield AFMS = Adafruit_MotorShield();

Adafruit_DCMotor *front_left = AFMS.getMotor(1);
Adafruit_DCMotor *front_right = AFMS.getMotor(2);
Adafruit_DCMotor *back_left = AFMS.getMotor(3);
Adafruit_DCMotor *back_right = AFMS.getMotor(4);

// Handles startup and shutdown of ROS
ros::NodeHandle nh;

////////////////// Tick Data Publishing Variables and Constants ///////////////

// Encoder output to Arduino Interrupt pin. Tracks the tick count.
#define ENC_IN_LEFT_A 11
#define ENC_IN_RIGHT_A 14
#define ENC_IN_BLEFT_A 21
#define ENC_IN_BRIGHT_A 8

// Other encoder output to Arduino to keep track of wheel direction
// Tracks the direction of rotation.
#define ENC_IN_LEFT_B 10
#define ENC_IN_RIGHT_B 15
#define ENC_IN_BLEFT_B 21
```

```
#define ENC_IN_BRIGHT_B 7

// True = Forward; False = Reverse
boolean Direction_left = true;
boolean Direction_right = true;

// Minumum and maximum values for 16-bit integers
// Range of 65,535
const int encoder_minimum = -32768;
const int encoder_maximum = 32767;

// Keep track of the number of wheel ticks
std_msgs::Int16 right_wheel_tick_count;
ros::Publisher rightPub("right_ticks", &right_wheel_tick_count);
std_msgs::Int16 bright_wheel_tick_count;
ros::Publisher brightPub("bright_ticks", &bright_wheel_tick_count);


std_msgs::Int16 left_wheel_tick_count;
ros::Publisher leftPub("left_ticks", &left_wheel_tick_count);
std_msgs::Int16 bleft_wheel_tick_count;
ros::Publisher bleftPub("bleft_ticks", &bleft_wheel_tick_count);
// Time interval for measurements in milliseconds
const int interval = 30;
long previousMillis = 0;
long currentMillis = 0;
// Increment the number of ticks
void right_wheel_tick() {

  // Read the value for the encoder for the right wheel
  int val = digitalRead(ENC_IN_RIGHT_B);

  if (val == LOW) {
    Direction_right = true; // Forward
  }
  else {
    Direction_right = false; // Reverse
  }

  if (Direction_right) {

    if (right_wheel_tick_count.data == encoder_maximum) {
      right_wheel_tick_count.data = encoder_minimum;
    }
    else {
      right_wheel_tick_count.data++;
```

```
    }
  }
  else {
    if (right_wheel_tick_count.data == encoder_minimum) {
      right_wheel_tick_count.data = encoder_maximum;
    }
    else {
      right_wheel_tick_count.data--;
    }
  }
}
void bright_wheel_tick() {

  // Read the value for the encoder for the right wheel
  int val = digitalRead(ENC_IN_BRIGHT_B);

  if (val == LOW) {
    Direction_right = false; // Forward
  }
  else {
    Direction_right = true; // Reverse
  }

  if (Direction_right) {

    if (bright_wheel_tick_count.data == encoder_maximum) {
      bright_wheel_tick_count.data = encoder_minimum;
    }
    else {
      bright_wheel_tick_count.data++;
    }
  }
  else {
    if (bright_wheel_tick_count.data == encoder_minimum) {
      bright_wheel_tick_count.data = encoder_maximum;
    }
    else {
      bright_wheel_tick_count.data--;
    }
  }
}

// Increment the number of ticks
void left_wheel_tick() {

  // Read the value for the encoder for the left wheel
```

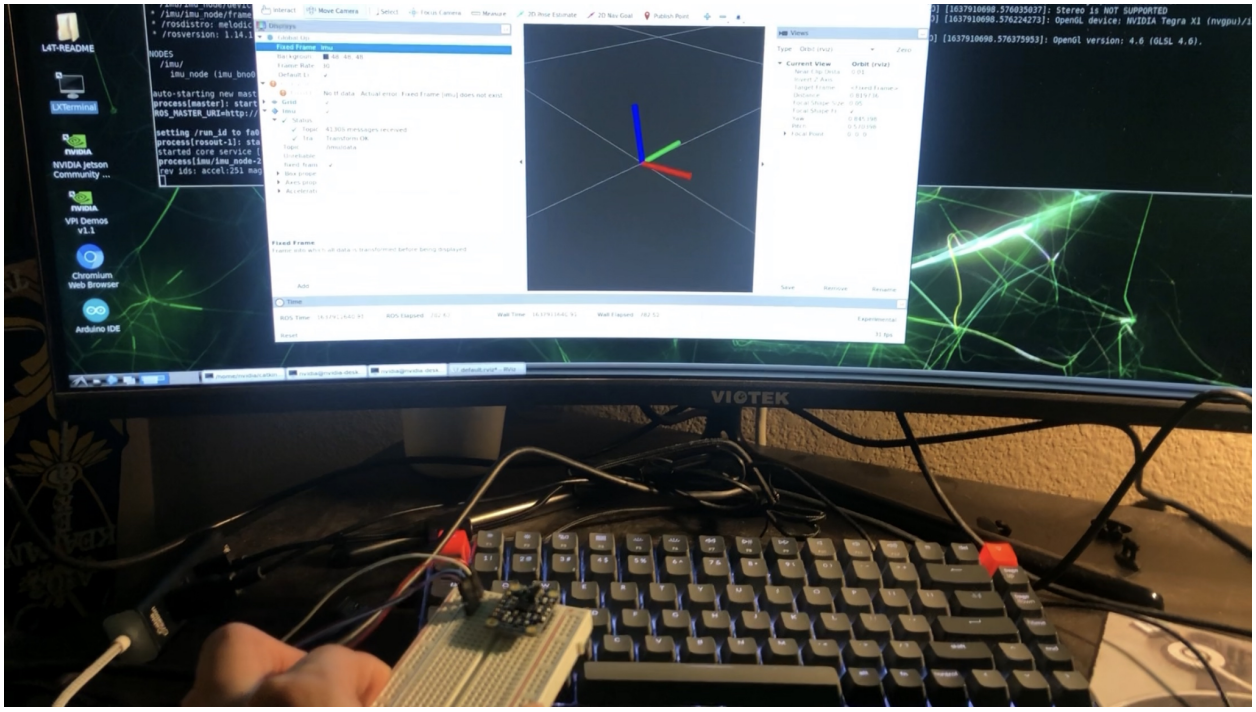
```
  int val = digitalRead(ENC_IN_LEFT_B);

  if (val == LOW) {
    Direction_left = true; // Reverse
  }
  else {
    Direction_left = false; // Forward
  }

  if (Direction_left) {
    if (left_wheel_tick_count.data == encoder_maximum) {
      left_wheel_tick_count.data = encoder_minimum;
    }
    else {
      left_wheel_tick_count.data++;
    }
  }
  else {
    if (left_wheel_tick_count.data == encoder_minimum) {
      left_wheel_tick_count.data = encoder_maximum;
    }
    else {
      left_wheel_tick_count.data--;
    }
  }
}
void bleft_wheel_tick() {

  // Read the value for the encoder for the left wheel
  int val = digitalRead(ENC_IN_BLEFT_B);

  if (val == LOW) {
    Direction_left = false; // Reverse
  }
  else {
    Direction_left = true; // Forward
  }

  if (Direction_left) {
    if (bleft_wheel_tick_count.data == encoder_maximum) {
      bleft_wheel_tick_count.data = encoder_minimum;
    }
    else {
      bleft_wheel_tick_count.data++;
    }
  }
```

```
  else {
    if (bleft_wheel_tick_count.data == encoder_minimum) {
      bleft_wheel_tick_count.data = encoder_maximum;
    }
    else {
      bleft_wheel_tick_count.data--;
    }
  }
}
void setup() {
  Serial.begin(57600);
  // Set pin states of the encoder
  pinMode(ENC_IN_LEFT_A , INPUT_PULLUP);
  pinMode(ENC_IN_LEFT_B , INPUT);
  pinMode(ENC_IN_RIGHT_A , INPUT_PULLUP);
  pinMode(ENC_IN_RIGHT_B , INPUT);
  pinMode(ENC_IN_BLEFT_A , INPUT_PULLUP);
  pinMode(ENC_IN_BLEFT_B , INPUT);
  pinMode(ENC_IN_BRIGHT_A , INPUT_PULLUP);
  pinMode(ENC_IN_BRIGHT_B , INPUT);

  // Every time the pin goes high, this is a tick
  attachInterrupt(digitalPinToInterrupt(ENC_IN_LEFT_A), left_wheel_tick, RISING);
  attachInterrupt(digitalPinToInterrupt(ENC_IN_RIGHT_A), right_wheel_tick, RISING);
  attachInterrupt(digitalPinToInterrupt(ENC_IN_BLEFT_A), bleft_wheel_tick, RISING);
  attachInterrupt(digitalPinToInterrupt(ENC_IN_BRIGHT_A), bright_wheel_tick, RISING);

  nh.getHardware()->setBaud(57600);
  nh.initNode();
  nh.advertise(rightPub);
  nh.advertise(leftPub);
  nh.advertise(brightPub);
  nh.advertise(bleftPub);
  nh.subscribe(subCmdVel);
}

void loop() {
  // Record the time
  currentMillis = millis();
  // If the time interval has passed, publish the number of ticks,
  // and calculate the velocities.
  if (currentMillis - previousMillis > interval) {
    previousMillis = currentMillis;
    // Publish tick counts to topics
    leftPub.publish( &left_wheel_tick_count );
    rightPub.publish( &right_wheel_tick_count );
```

```
    leftPub.publish( &bleft_wheel_tick_count );
    rightPub.publish( &bright_wheel_tick_count );
  }

  nh.spinOnce();
}
```

3. Task 2 - IMU visualizer and data publisher through ROS on Jetson Nano

Visualized and installed using BNO055 ROS package:
https://github.com/dheera/ros-imu-bno055.git



Added only the ROS node to the launch file.

```
  <node ns="imu" name="imu_node" pkg="imu_bno055" type="bno055_i2c_node"
respawn="true" respawn_delay="2">
    <param name="device" type="string" value="/dev/i2c-1"/>
    <param name="address" type="int" value="40"/>
    <param name="frame_id" type="string" value="imu"/>
  </node>
```

4. Task 3 - Motor Control through ROS on Jetson Nano, Teensy using cmd_vel and rqt_robot_steering

Code modified from [1] to utilize Adafruit Motor Shield and 4 motor drivers instead of 2. Constants are modified to better suit the motor drivers and motor shield.
Code uploaded to Teensy:

```
#include <ros.h>
#include <std_msgs/Int16.h>
#include <geometry_msgs/Twist.h>
#include <Wire.h>
#include <Adafruit_MotorShield.h>

Adafruit_MotorShield AFMS = Adafruit_MotorShield();

Adafruit_DCMotor *front_left = AFMS.getMotor(1);
Adafruit_DCMotor *front_right = AFMS.getMotor(2);
Adafruit_DCMotor *back_left = AFMS.getMotor(3);
Adafruit_DCMotor *back_right = AFMS.getMotor(4);

// Handles startup and shutdown of ROS
ros::NodeHandle nh;

////////////////// Tick Data Publishing Variables and Constants ///////////////

// Encoder output to Arduino Interrupt pin. Tracks the tick count.
#define ENC_IN_LEFT_A 11
#define ENC_IN_RIGHT_A 14
#define ENC_IN_BLEFT_A 21
#define ENC_IN_BRIGHT_A 8

// Other encoder output to Arduino to keep track of wheel direction
// Tracks the direction of rotation.
#define ENC_IN_LEFT_B 10
#define ENC_IN_RIGHT_B 15
#define ENC_IN_BLEFT_B 21
#define ENC_IN_BRIGHT_B 7

// True = Forward; False = Reverse
boolean Direction_left = true;
boolean Direction_right = true;

// Minumum and maximum values for 16-bit integers
// Range of 65,535
const int encoder_minimum = -32768;
const int encoder_maximum = 32767;

// Keep track of the number of wheel ticks
```

```
std_msgs::Int16 right_wheel_tick_count;
ros::Publisher rightPub("right_ticks", &right_wheel_tick_count);
std_msgs::Int16 bright_wheel_tick_count;
ros::Publisher brightPub("bright_ticks", &bright_wheel_tick_count);


std_msgs::Int16 left_wheel_tick_count;
ros::Publisher leftPub("left_ticks", &left_wheel_tick_count);
std_msgs::Int16 bleft_wheel_tick_count;
ros::Publisher bleftPub("bleft_ticks", &bleft_wheel_tick_count);

// Time interval for measurements in milliseconds
const int interval = 30;
long previousMillis = 0;
long currentMillis = 0;

////////////////// Motor Controller Variables and Constants ////////////////////

// How much the PWM value can change each cycle
const int PWM_INCREMENT = 10;

// Number of ticks per wheel revolution. We won't use this in this code.
// Wheel radius in meters
// Distance from center of the left tire to the center of the right tire in m
const int TICKS_PER_REVOLUTION = 135;
const double WHEEL_RADIUS = 0.033;
const double WHEEL_BASE = 0.15;

// Number of ticks a wheel makes moving a linear distance of 1 meter
// This value was measured manually.
//const double TICKS_PER_METER = 3100; // Originally 2880
const double TICKS_PER_METER = 3100; // Originally 2880

// Proportional constant, which was measured by measuring the
// PWM-Linear Velocity relationship for the robot.
const int K_P = 278;
//const int K_P = 1;

// Y-intercept for the PWM-Linear Velocity relationship for the robot
const int b = 52;

// Correction multiplier for drift. Chosen through experimentation.
const int DRIFT_MULTIPLIER = 120;
//const int DRIFT_MULTIPLIER = 1;

// Turning PWM output (0 = min, 255 = max for PWM values)
```

```
const int PWM_TURN = 40;

// Set maximum and minimum limits for the PWM values
const int PWM_MIN = 80; // about 0.1 m/s
const int PWM_MAX = 255; // about 0.172 m/s

// Set linear velocity and PWM variable values for each wheel
double velLeftWheel = 0;
double velRightWheel = 0;
double velbLeftWheel = 0;
double velbRightWheel = 0;
double pwmLeftReq = 0;
double pwmRightReq = 0;
double pwmbLeftReq = 0;
double pwmbRightReq = 0;

// Record the time that the last velocity command was received
double lastCmdVelReceived = 0;

////////////////////// Tick Data Publishing Functions ////////////////////////

// Increment the number of ticks
void right_wheel_tick() {

  // Read the value for the encoder for the right wheel
  int val = digitalRead(ENC_IN_RIGHT_B);

  if (val == LOW) {
    Direction_right = true; // Forward
  }
  else {
    Direction_right = false; // Reverse
  }

  if (Direction_right) {

    if (right_wheel_tick_count.data == encoder_maximum) {
      right_wheel_tick_count.data = encoder_minimum;
    }
    else {
      right_wheel_tick_count.data++;
    }
  }
  else {
    if (right_wheel_tick_count.data == encoder_minimum) {
      right_wheel_tick_count.data = encoder_maximum;
```

```
    }
    else {
      right_wheel_tick_count.data--;
    }
  }
}
void bright_wheel_tick() {

  // Read the value for the encoder for the right wheel
  int val = digitalRead(ENC_IN_BRIGHT_B);

  if (val == LOW) {
    Direction_right = false; // Forward
  }
  else {
    Direction_right = true; // Reverse
  }

  if (Direction_right) {

    if (bright_wheel_tick_count.data == encoder_maximum) {
      bright_wheel_tick_count.data = encoder_minimum;
    }
    else {
      bright_wheel_tick_count.data++;
    }
  }
  else {
    if (bright_wheel_tick_count.data == encoder_minimum) {
      bright_wheel_tick_count.data = encoder_maximum;
    }
    else {
      bright_wheel_tick_count.data--;
    }
  }
}

// Increment the number of ticks
void left_wheel_tick() {

  // Read the value for the encoder for the left wheel
  int val = digitalRead(ENC_IN_LEFT_B);

  if (val == LOW) {
    Direction_left = true; // Reverse
  }
```

```
  else {
    Direction_left = false; // Forward
  }

  if (Direction_left) {
    if (left_wheel_tick_count.data == encoder_maximum) {
      left_wheel_tick_count.data = encoder_minimum;
    }
    else {
      left_wheel_tick_count.data++;
    }
  }
  else {
    if (left_wheel_tick_count.data == encoder_minimum) {
      left_wheel_tick_count.data = encoder_maximum;
    }
    else {
      left_wheel_tick_count.data--;
    }
  }
}
void bleft_wheel_tick() {

  // Read the value for the encoder for the left wheel
  int val = digitalRead(ENC_IN_BLEFT_B);

  if (val == LOW) {
    Direction_left = false; // Reverse
  }
  else {
    Direction_left = true; // Forward
  }

  if (Direction_left) {
    if (bleft_wheel_tick_count.data == encoder_maximum) {
      bleft_wheel_tick_count.data = encoder_minimum;
    }
    else {
      bleft_wheel_tick_count.data++;
    }
  }
  else {
    if (bleft_wheel_tick_count.data == encoder_minimum) {
      bleft_wheel_tick_count.data = encoder_maximum;
    }
    else {
```

```
      bleft_wheel_tick_count.data--;
    }
  }
}
////////////////////// Motor Controller Functions ///////////////////////////

// Calculate the left wheel linear velocity in m/s every time a
// tick count message is rpublished on the /left_ticks topic.
void calc_vel_left_wheel(){

  // Previous timestamp
  static double prevTime = 0;

  // Variable gets created and initialized the first time a function is called.
  static int prevLeftCount = 0;

  // Manage rollover and rollunder when we get outside the 16-bit integer range
  int numOfTicks = (65535 + left_wheel_tick_count.data - prevLeftCount) % 65535;

  // If we have had a big jump, it means the tick count has rolled over.
  if (numOfTicks > 10000) {
        numOfTicks = 0 - (65535 - numOfTicks);
  }

  // Calculate wheel velocity in meters per second
  velLeftWheel = numOfTicks/TICKS_PER_METER/((millis()/1000)-prevTime);

  // Keep track of the previous tick count
  prevLeftCount = left_wheel_tick_count.data;

  // Update the timestamp
  prevTime = (millis()/1000);

}
void calc_vel_bleft_wheel(){

  // Previous timestamp
  static double prevTime = 0;

  // Variable gets created and initialized the first time a function is called.
  static int prevbLeftCount = 0;

  // Manage rollover and rollunder when we get outside the 16-bit integer range
  int numOfTicks = (65535 + bleft_wheel_tick_count.data - prevbLeftCount) % 65535;

  // If we have had a big jump, it means the tick count has rolled over.
```

```
  if (numOfTicks > 10000) {
        numOfTicks = 0 - (65535 - numOfTicks);
  }

  // Calculate wheel velocity in meters per second
  velLeftWheel = numOfTicks/TICKS_PER_METER/((millis()/1000)-prevTime);

  // Keep track of the previous tick count
  prevbLeftCount = bleft_wheel_tick_count.data;

  // Update the timestamp
  prevTime = (millis()/1000);

}

// Calculate the right wheel linear velocity in m/s every time a
// tick count message is published on the /right_ticks topic.
void calc_vel_right_wheel(){

  // Previous timestamp
  static double prevTime = 0;

  // Variable gets created and initialized the first time a function is called.
  static int prevRightCount = 0;

  // Manage rollover and rollunder when we get outside the 16-bit integer range
  int numOfTicks = (65535 + right_wheel_tick_count.data - prevRightCount) % 65535;

  if (numOfTicks > 10000) {
        numOfTicks = 0 - (65535 - numOfTicks);
  }

  // Calculate wheel velocity in meters per second
  velRightWheel = numOfTicks/TICKS_PER_METER/((millis()/1000)-prevTime);

  prevRightCount = right_wheel_tick_count.data;

  prevTime = (millis()/1000);

}
void calc_vel_bright_wheel(){

  // Previous timestamp
  static double prevTime = 0;

  // Variable gets created and initialized the first time a function is called.
```

```
  static int prevbRightCount = 0;

  // Manage rollover and rollunder when we get outside the 16-bit integer range
  int numOfTicks = (65535 + bright_wheel_tick_count.data - prevbRightCount) % 65535;

  if (numOfTicks > 10000) {
        numOfTicks = 0 - (65535 - numOfTicks);
  }

  // Calculate wheel velocity in meters per second
  velRightWheel = numOfTicks/TICKS_PER_METER/((millis()/1000)-prevTime);

  prevbRightCount = bright_wheel_tick_count.data;

  prevTime = (millis()/1000);

}

// Take the velocity command as input and calculate the PWM values.
void calc_pwm_values(const geometry_msgs::Twist& cmdVel) {

  // Record timestamp of last velocity command received
  lastCmdVelReceived = (millis()/1000);

  // Calculate the PWM value given the desired velocity
  pwmLeftReq = K_P * cmdVel.linear.x + b;
  pwmRightReq = K_P * cmdVel.linear.x + b;
  pwmbLeftReq = K_P * cmdVel.linear.x + b;
  pwmbRightReq = K_P * cmdVel.linear.x + b;

  // Check if we need to turn
  if (cmdVel.angular.z != 0.0) {

    // Turn left
    if (cmdVel.angular.z > 0.0) {
      pwmLeftReq = -PWM_TURN;
      pwmbLeftReq = -PWM_TURN;
      pwmRightReq = PWM_TURN;
      pwmbRightReq = PWM_TURN;
    }
    // Turn right
    else {
      pwmLeftReq = PWM_TURN;
      pwmbLeftReq = PWM_TURN;
      pwmRightReq = -PWM_TURN;
      pwmbRightReq = -PWM_TURN;
```

```
    }
  }
  // Go straight
  else {

    // Remove any differences in wheel velocities
    // to make sure the robot goes straight
    static double prevDiff = 0;
    static double prevPrevDiff = 0;
    double currDifference = velLeftWheel - velRightWheel;
    double avgDifference = (prevDiff+prevPrevDiff+currDifference)/3;
    prevPrevDiff = prevDiff;
    prevDiff = currDifference;

    // Correct PWM values of both wheels to make the vehicle go straight
    pwmLeftReq -= (int)(avgDifference * DRIFT_MULTIPLIER);
    pwmRightReq += (int)(avgDifference * DRIFT_MULTIPLIER);
    pwmbLeftReq -= (int)(avgDifference * DRIFT_MULTIPLIER);
    pwmbRightReq += (int)(avgDifference * DRIFT_MULTIPLIER);
  }

  // Handle low PWM values
/*   if (abs(pwmLeftReq) < PWM_MIN) {
    pwmLeftReq = 0;
  }
  if (abs(pwmRightReq) < PWM_MIN) {
    pwmRightReq = 0;
  }
  if (abs(pwmbLeftReq) < PWM_MIN) {
    pwmbLeftReq = 0;
  }
  if (abs(pwmbRightReq) < PWM_MIN) {
    pwmbRightReq = 0;
  }   */
}

void set_pwm_values() {

  // These variables will hold our desired PWM values
  static int pwmLeftOut = 0;
  static int pwmRightOut = 0;

  // If the required PWM is of opposite sign as the output PWM, we want to
  // stop the car before switching direction
  static bool stopped = false;
  if ((pwmLeftReq * velLeftWheel < 0 && pwmLeftOut != 0) ||
```

```
     (pwmRightReq * velRightWheel < 0 && pwmRightOut != 0)) {
  pwmLeftReq = 0;
  pwmRightReq = 0;
}

// Set the direction of the motors
if (pwmLeftReq > 0) { // Left wheel forward
  // digitalWrite(in1, HIGH);
  // digitalWrite(in2, LOW);
  back_left->run(FORWARD);
  front_left->run(FORWARD);
}
else if (pwmLeftReq < 0) { // Left wheel reverse
  // digitalWrite(in1, LOW);
  // digitalWrite(in2, HIGH);
  back_left->run(BACKWARD);
  front_left->run(BACKWARD);
}
else if (pwmLeftReq == 0 && pwmLeftOut == 0 ) { // Left wheel stop
  // digitalWrite(in1, LOW);
  // digitalWrite(in2, LOW);
  back_left->run(RELEASE);
  front_left->run(RELEASE);
}
else { // Left wheel stop
  // digitalWrite(in1, LOW);
  // digitalWrite(in2, LOW);
  back_left->run(RELEASE);
  front_left->run(RELEASE);
}

if (pwmRightReq > 0) { // Right wheel forward
  // digitalWrite(in3, HIGH);
  // digitalWrite(in4, LOW);
  back_right->run(FORWARD);
  front_right->run(FORWARD);
}
else if(pwmRightReq < 0) { // Right wheel reverse
  // digitalWrite(in3, LOW);
  // digitalWrite(in4, HIGH);
  back_right->run(BACKWARD);
  front_right->run(BACKWARD);
}
else if (pwmRightReq == 0 && pwmRightOut == 0) { // Right wheel stop
  // digitalWrite(in3, LOW);
  // digitalWrite(in4, LOW);
```

```
    back_right->run(RELEASE);
    front_right->run(RELEASE);
  }
  else { // Right wheel stop
    // digitalWrite(in3, LOW);
    // digitalWrite(in4, LOW);
    back_right->run(RELEASE);
    front_right->run(RELEASE);
  }

  // Increase the required PWM if the robot is not moving
  if (pwmLeftReq != 0 && velLeftWheel == 0) {
    pwmLeftReq *= 1.5;
  }
  if (pwmRightReq != 0 && velRightWheel == 0) {
    pwmRightReq *= 1.5;
  }
  /* if (pwmbLeftReq != 0 && velbLeftWheel == 0) {
    pwmbLeftReq *= 1.5;
  }
  if (pwmbRightReq != 0 && velbRightWheel == 0) {
    pwmbRightReq *= 1.5;
  } */

  // Calculate the output PWM value by making slow changes to the current value
  if (abs(pwmLeftReq) > pwmLeftOut) {
    pwmLeftOut += PWM_INCREMENT;
  }
  else if (abs(pwmLeftReq) < pwmLeftOut) {
    pwmLeftOut -= PWM_INCREMENT;
  }
  else{}

  if (abs(pwmRightReq) > pwmRightOut) {
    pwmRightOut += PWM_INCREMENT;
  }
  else if(abs(pwmRightReq) < pwmRightOut) {
    pwmRightOut -= PWM_INCREMENT;
  }
  else{}

  // Conditional operator to limit PWM output at the maximum
  pwmLeftOut = (pwmLeftOut > PWM_MAX) ? PWM_MAX : pwmLeftOut;
  pwmRightOut = (pwmRightOut > PWM_MAX) ? PWM_MAX : pwmRightOut;

  // PWM output cannot be less than 0
```

```
  pwmLeftOut = (pwmLeftOut < 0) ? 0 : pwmLeftOut;
  pwmRightOut = (pwmRightOut < 0) ? 0 : pwmRightOut;

  // Set the PWM value on the pins
  // analogWrite(enA, pwmLeftOut);
  // analogWrite(enB, pwmRightOut);
  front_left->setSpeed(pwmLeftOut);
  back_left->setSpeed(pwmLeftOut);
  front_right->setSpeed(pwmRightOut);
  back_right->setSpeed(pwmRightOut);
}

// Set up ROS subscriber to the velocity command
ros::Subscriber<geometry_msgs::Twist> subCmdVel("cmd_vel", &calc_pwm_values );

void setup() {
  Serial.begin(57600);
  // Set pin states of the encoder
  pinMode(ENC_IN_LEFT_A , INPUT_PULLUP);
  pinMode(ENC_IN_LEFT_B , INPUT);
  pinMode(ENC_IN_RIGHT_A , INPUT_PULLUP);
  pinMode(ENC_IN_RIGHT_B , INPUT);
  pinMode(ENC_IN_BLEFT_A , INPUT_PULLUP);
  pinMode(ENC_IN_BLEFT_B , INPUT);
  pinMode(ENC_IN_BRIGHT_A , INPUT_PULLUP);
  pinMode(ENC_IN_BRIGHT_B , INPUT);

  // Every time the pin goes high, this is a tick
  attachInterrupt(digitalPinToInterrupt(ENC_IN_LEFT_A), left_wheel_tick, RISING);
  attachInterrupt(digitalPinToInterrupt(ENC_IN_RIGHT_A), right_wheel_tick, RISING);
  attachInterrupt(digitalPinToInterrupt(ENC_IN_BLEFT_A), bleft_wheel_tick, RISING);
  attachInterrupt(digitalPinToInterrupt(ENC_IN_BRIGHT_A), bright_wheel_tick, RISING);

  // Motor control pins are outputs
  // pinMode(enA, OUTPUT);
  // pinMode(enB, OUTPUT);
  // pinMode(in1, OUTPUT);
  // pinMode(in2, OUTPUT);
  // pinMode(in3, OUTPUT);
  // pinMode(in4, OUTPUT);

  AFMS.begin();
  // Turn off motors - Initial state
  // digitalWrite(in1, LOW);
  // digitalWrite(in2, LOW);
  // digitalWrite(in3, LOW);
```

```
  // digitalWrite(in4, LOW);
  front_right->run(RELEASE);
  front_left->run(RELEASE);
  back_right->run(RELEASE);
  back_left->run(RELEASE);

  // Set the motor speed
  // analogWrite(enA, 0);
  // analogWrite(enB, 0);
  front_right->setSpeed(0);
  front_left->setSpeed(0);
  back_right->setSpeed(0);
  back_left->setSpeed(0);

  // ROS Setup
  //nh.getHardware()->setBaud(115200);
  nh.getHardware()->setBaud(57600);
  nh.initNode();
  nh.advertise(rightPub);
  nh.advertise(leftPub);
  nh.advertise(brightPub);
  nh.advertise(bleftPub);
  nh.subscribe(subCmdVel);
}

void loop() {


  // Record the time
  currentMillis = millis();

  // If the time interval has passed, publish the number of ticks,
  // and calculate the velocities.
  if (currentMillis - previousMillis > interval) {

    previousMillis = currentMillis;

    // Publish tick counts to topics
    leftPub.publish( &left_wheel_tick_count );
    rightPub.publish( &right_wheel_tick_count );
    leftPub.publish( &bleft_wheel_tick_count );
    rightPub.publish( &bright_wheel_tick_count );

    // Calculate the velocity of the right and left wheels
    calc_vel_right_wheel();
    calc_vel_left_wheel();
```

```
    calc_vel_bright_wheel();
    calc_vel_bleft_wheel();

  }

  // Stop the car if there are no cmd_vel messages
  if((millis()/1000) - lastCmdVelReceived > 1) {
    pwmLeftReq = 0;
    pwmRightReq = 0;
  }

  set_pwm_values();
  nh.spinOnce();
}
```

5. **Task 4 - Navigation Stack**

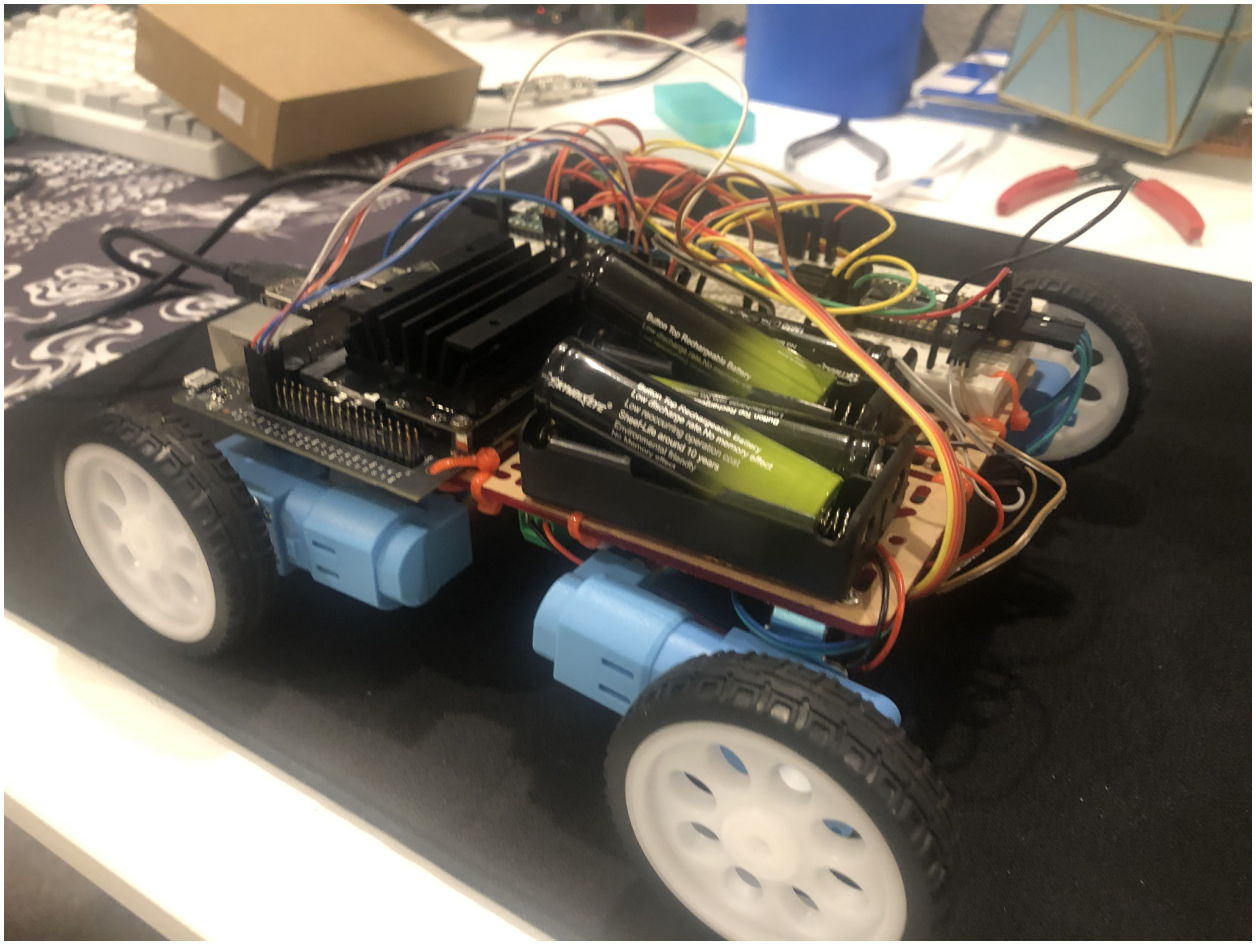
Installed Robot Pose EKF for ROS Melodic. Subscribes to wheel encoder information (odometry) and IMU data to be published as localization data.

Launch file was updated to:

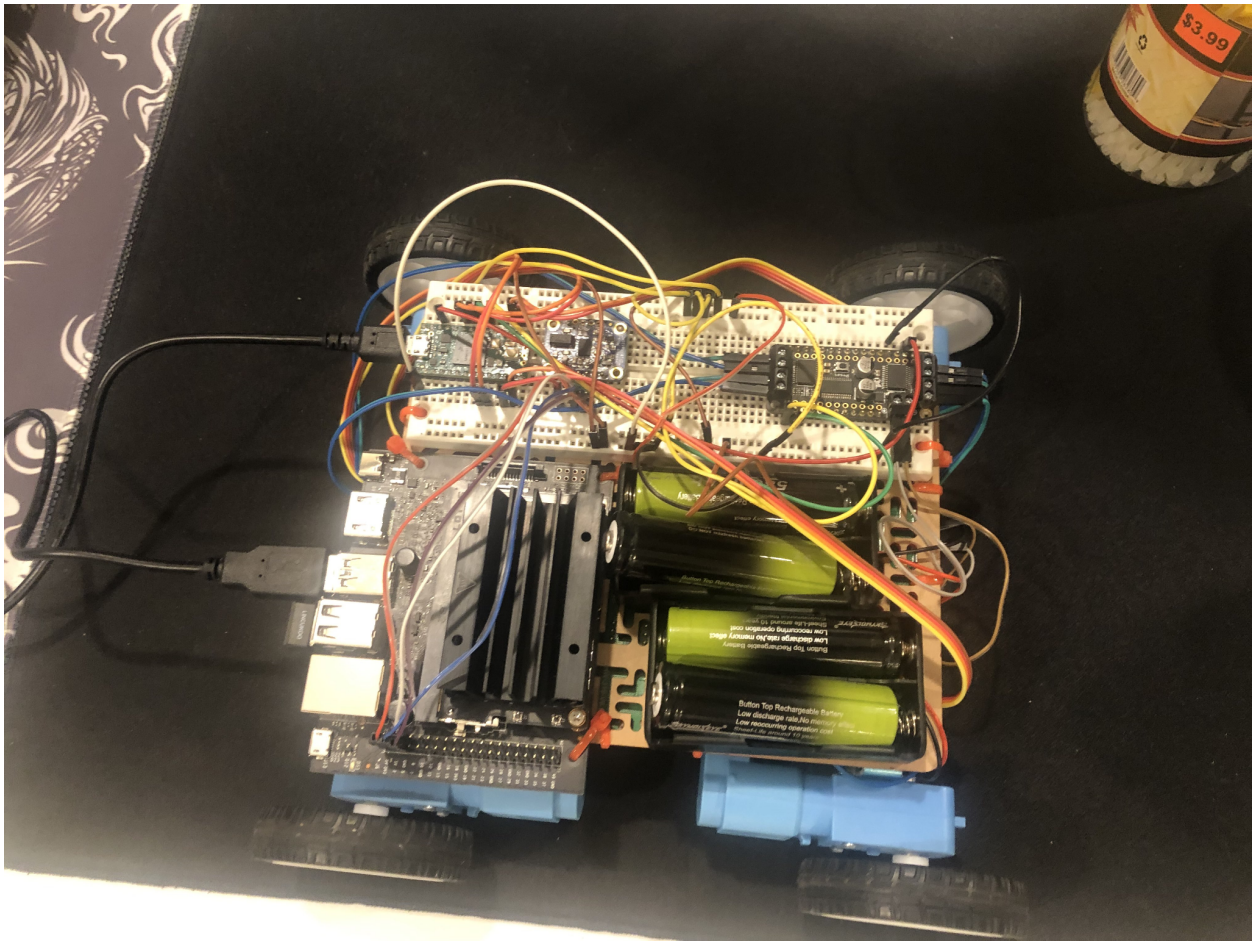
```
<launch>
  <node pkg="rosserial_python" type="serial_node.py" name="serial_node">
    <param name="port" value="/dev/ttyACM0"/>
    <param name="baud" value="57600"/>
  </node>

  <node ns="imu" name="imu_node" pkg="imu_bno055" type="bno055_i2c_node"
respawn="true" respawn_delay="2">
    <param name="device" type="string" value="/dev/i2c-1"/>
    <param name="address" type="int" value="40"/>
    <param name="frame_id" type="string" value="imu"/>
  </node>

  <node pkg="rqt_robot_steering" type="rqt_robot_steering"
name="rqt_robot_steering">
  </node>
  <!-- Extended Kalman Filter from robot_pose_ekf Node-->
```

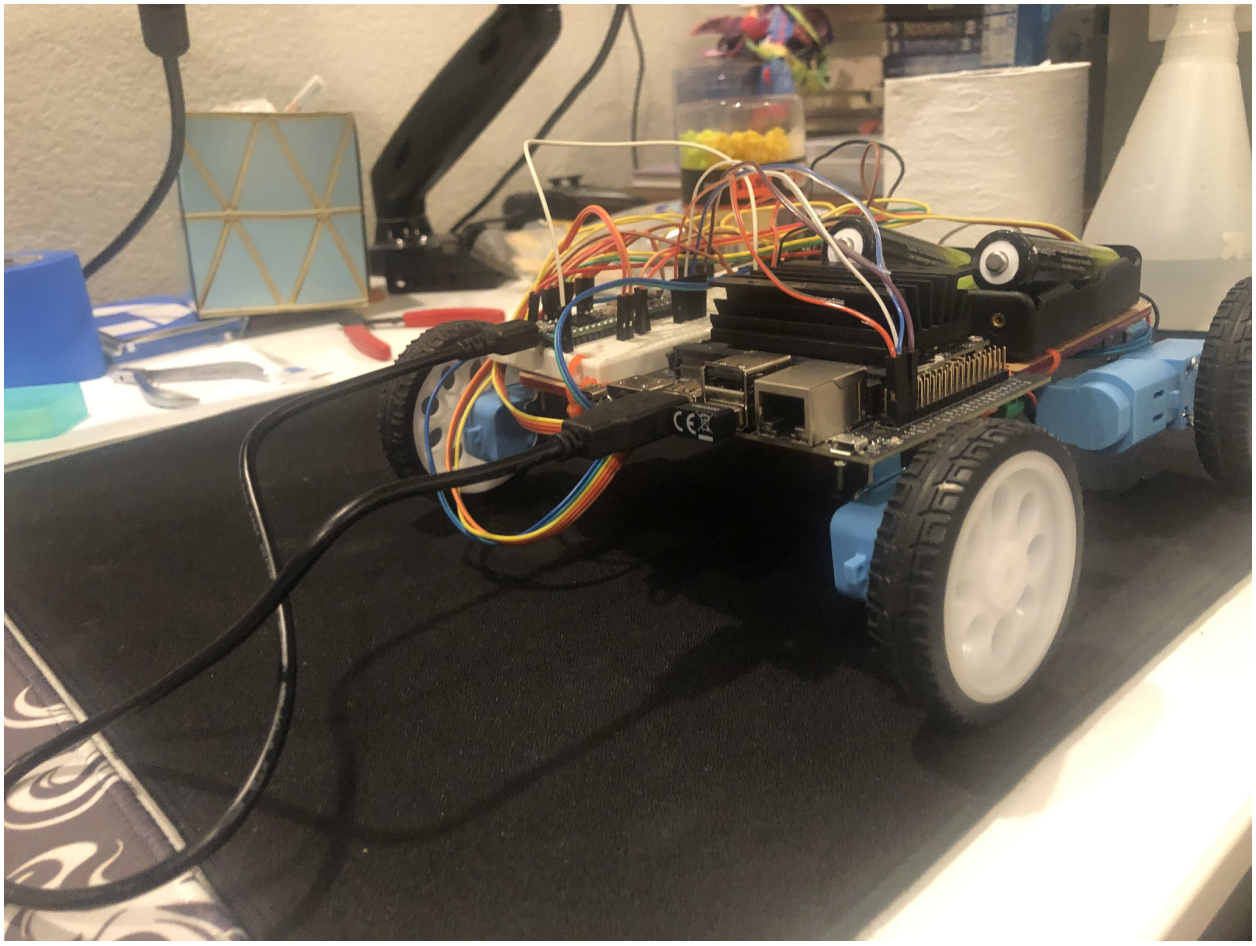
```
  <!-- Subscribe: /odom, /imu_data, /vo -->
  <!-- Publish: /robot_pose_ekf/odom_combined -->
  <remap from="odom" to="odom_data_quat" />
  <remap from="imu_data" to="imu/data" />
  <node pkg="robot_pose_ekf" type="robot_pose_ekf"
name="robot_pose_ekf">
    <param name="output_frame" value="odom"/>
    <param name="base_footprint_frame" value="base_footprint"/>
    <param name="freq" value="30.0"/>
    <param name="sensor_timeout" value="1.0"/>
    <param name="odom_used" value="true"/>
    <param name="imu_used" value="true"/>
    <param name="vo_used" value="false"/>
    <param name="gps_used" value="false"/>
    <param name="debug" value="false"/>
    <param name="self_diagnose" value="false"/>
  </node>

  <!-- Initial Pose and Goal Publisher -->
  <!-- Publish: /initialpose, /move_base_simple/goal -->
  <node pkg="rviz" type="rviz" name="rviz">
  </node>
  <!-- Subscribe: /initialpose, /move_base_simple/goal -->
  <!-- Publish: /initial_2d, /goal_2d -->
  <node pkg="localization_data_pub" type="rviz_click_to_2d"
name="rviz_click_to_2d">
  </node>

  <!-- Wheel Odometry Publisher -->
  <!-- Subscribe: /right_ticks, /left_ticks, /initial_2d -->
  <!-- Publish: /odom_data_euler, /odom_data_quat -->
  <node pkg="localization_data_pub" type="ekf_odom_pub"
name="ekf_odom_pub">
  </node>
</launch>
```

6. **Jetbot Set Up**



CPE467 **Midterm 2** Keanan Gabertan
Fall 2021 Vincent Vuong





7. **Video Link to demo**
   https://youtu.be/CbMG2iIXGFM
8. **Github link to the shared repository**
   https://github.com/EyeEsquire/cpe476.git