Mechatronics Final Project Report

Group Members:

- Akshay Singaram Tamizhselvam
- Shangdong Yang
- Taiyi Pan
- Archit Vireshkumar Sharma

ROB-GY 5103 Mechatronics

12/15/2021

Contents

1. INTRODUCTION

1.1 Problem Statement

Traffic volume passing through any particular crossroad is very dynamic and can drastically vary at different times of the day. In order to regulate the traffic flow, traffic lights are used which can automatically direct traffic flow. This maintains order in what would otherwise be a chaotic flow of traffic via a busy crossroad junction. Although traffic lights follow a predetermined pattern of changing lights, determined by an algorithm, the lights can be ineffective in certain situations. Quite often, when roads are being repaired, or an accident has occurred on a road, traffic is redirected. This can result in far more traffic flowing through a junction than the junction was originally designed to accommodate. As a result, the traffic light at that particular intersection becomes unable to maintain a smooth traffic flow. In such situations, a traffic police officer steps in to manually redirect traffic to prevent/dislodge traffic jams.

Traffic police officers use hand gestures, often while holding a baton, to signal when a driver can proceed to cross the intersection. Although this is effective to navigate the front rows of traffic, drivers further behind the line can find it difficult to see what is going on. This often results in impatient drivers making maneuvers which may be dangerous and/or disrupt the flow of traffic. This project aims to design a solution to counteract the issue of drivers not having proper view of the traffic police officer.

1.2 Objectives

- Device a solution for all drivers to have visual cues to know when to move as directed by the traffic police.
- Make the solution easy to use and operate by the traffic police officer.
- Design the solution to be portable and have a small form factor.

2. METHODOLOGY

2.1 Solution

The solution to tackle the problem was to design a hand mounted device that can read a police officer's hand signals. The device would detect when the police officer is indicating "stop" and "go" with their hands as shown in Figure 1.

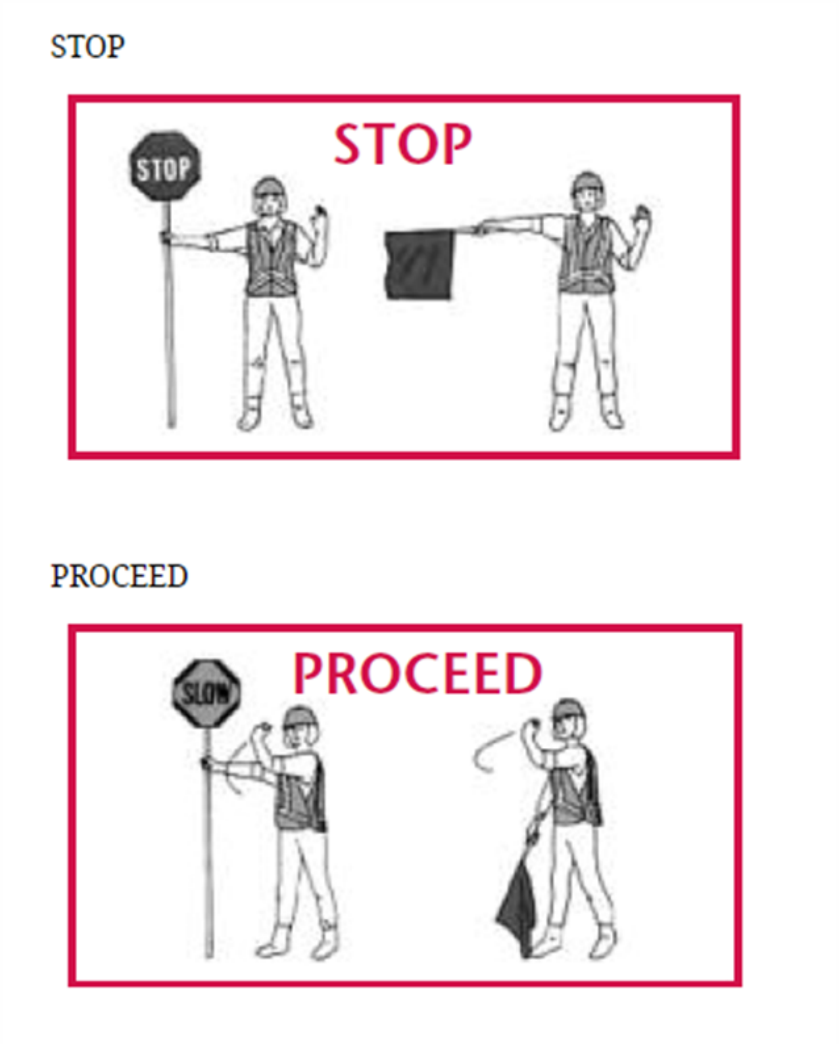


Figure 1: Hand gestures to control traffic (Department of Motor Vehicles)

Data recorded by the device mounted on the hand will be sent to a distant controller which controls the traffic lights. When the device detects the “STOP” gesture as seen in Figure 1, the traffic light for the corresponding lane will turn red. Similarly, on detecting green, the traffic light will turn green. As a result, the traffic police will be able to control traffic lights using hand gestures.

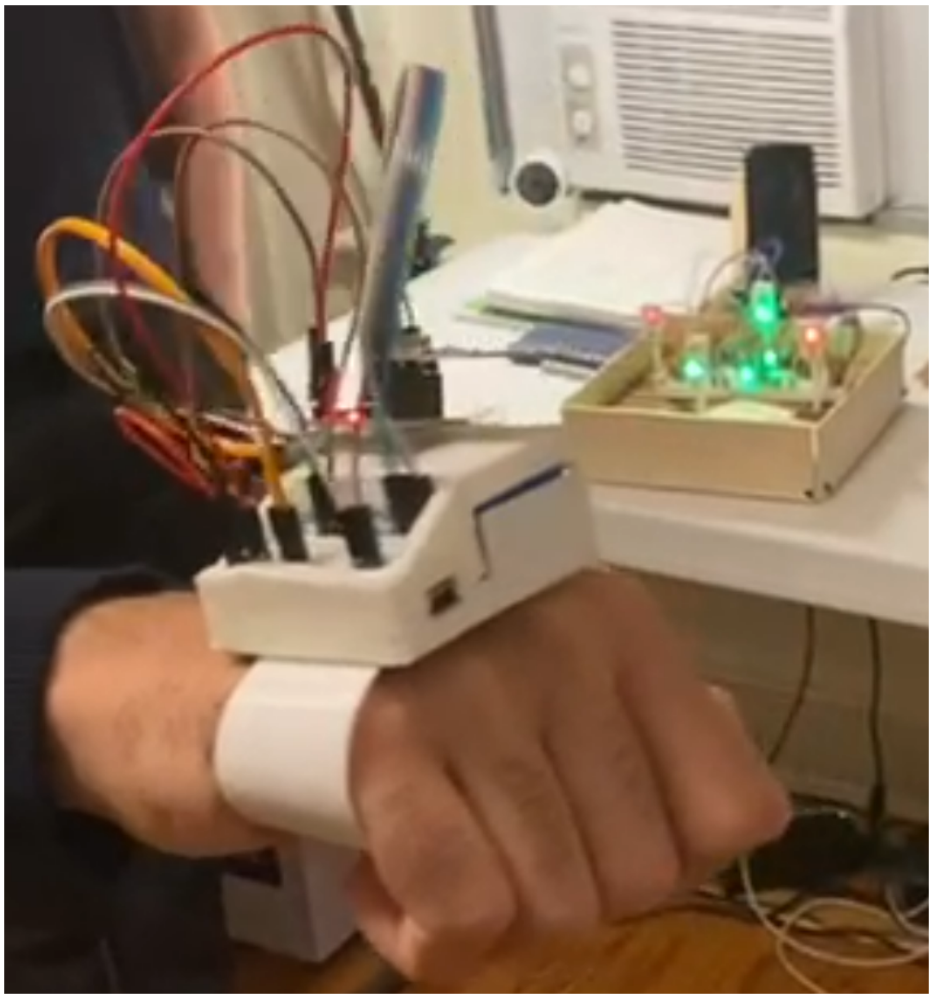


Figure 2: Transmitter (on hand) and Receiver (controlling traffic lights)

2.2 Components

Serial Number	Components	Description	Quantities
1	Arduino Nano	MCU on the hand	1
2	Arduino Uno	MCU controlling lights	1
3	MPU6050	Accelerometer detecting hand gestures	1
4	NRF24L01	Communication module	2
5	Jumpers	Connecting wires	NA
6	LEDs	Traffic lights	12
7	Buttons	Manual/Auto and Calibration button	2

2.3 Component Selection Criteria

1. NRF Module:

   The main reason why we chose radio frequency over other communication protocols is due to its simplicity and modularity. When it comes to the NRF module it sends data as radio frequency with an address attached to each message. When another NRF picks up the same radio signal, it tries to decode starting with the address and only if the address matches with the one defined for it will it decode the message. This makes it very simple to use for prototyping.

   The drawback of this is that it can be hacked easily as there isn't any particular firewall to breach.

2. MPU6050:

   It is the cheapest, low power accelerometer on the market. It has a lot of libraries written for Arduino, Raspberry Pi, and other microcontrollers. It can sense acceleration and gyroscope values very efficiently. The reason we used MPU6050 and not a camera is because using a camera is computationally expensive as it would need a processor like Raspberry Pi to be running. So it’s better to use a cheap sensor as we only need the hand gestures.

   The drawback of this sensor is that while integrating the values of the MPU we end up summing up the error terms as well which tends to get accumulated during integration.

3. Arduino Nano and Uno:

We used Arduino Nano as it's small enough to fit on the hand and can work well with the glove. For controlling the lights we used an Uno instead of BS2 because BS2 doesn’t have SPI communication. It can only do serial communication and through this we can only interface it with a bluetooth module. A bluetooth module wouldn't be an ideal communication protocol here because the police officer would have to connect to the bluetooth device every time. The range of a bluetooth module is also comparatively shorter. Hence to avoid all these issues, we used Arduino Uno.

2.4 Circuitry

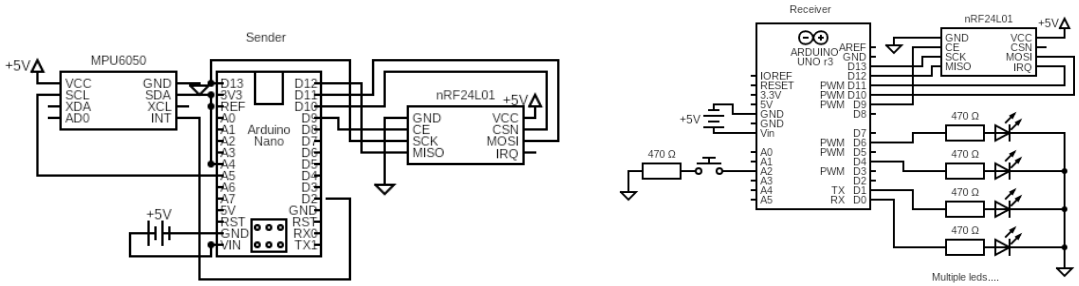


Figure 1: Schematic Diagram

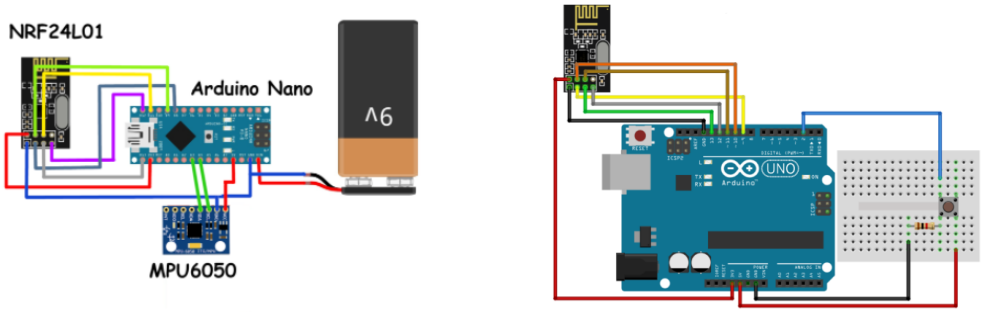
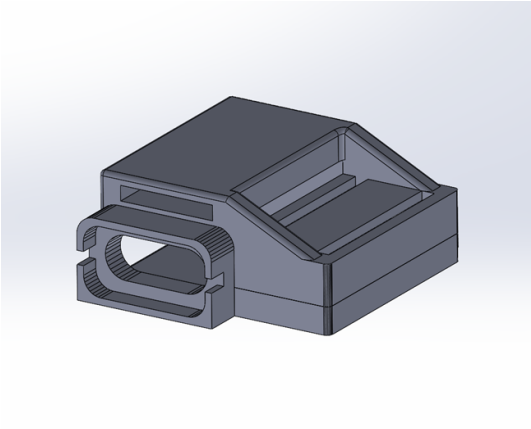
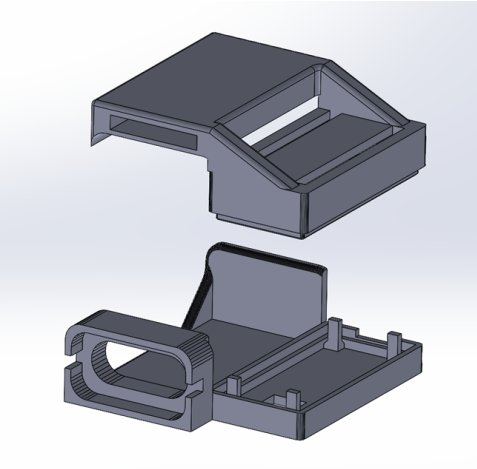


Figure 2: Illustrated Diagram (Arduino Project Hub, 2018)

2.5 Mechanical Design

In order to mount the MPU6050 sensor, NRF radio module, Arduino Nano, and a 9V battery onto the hand in correct orientation, a mounting case was designed using Solidworks and was 3D printed using TPU-95A filament. TPU-95A was used to allow for compliance as the case will be able to deform to contour the hand. The CAD model can be seen in figure below.





2.6 Mathematical Background

To indicate the gestures of a police officer, we need to track the motions of his hand. We used an Inertial Measurement Unit (IMU) – MPU6050 to realize this feature.

MPU6050 is the world's first integrated 6-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and a Digital Motion Processor™ (DMP) all in a small integrated circuit. And it is based on the I2C communication protocol, so we can just use 2 lines of code to get the raw data from MPU6050 to Arduino.

The gyroscope of MPU6050 measures rotational velocity or rate of change of the angular position over time, along the X, Y and Z axis. It uses MEMS technology and the Coriolis effect for measurement. The output of the gyroscope is in degrees per second, so in order to get the angular positions we just need to integrate the angular velocity.

```
// Currently the raw values are in degrees per seconds, deg/s, so we need to multiply
by sendonds (s) to get the angle in degrees

 gyroAngleX = gyroAngleX + GyroX * elapsedTime; // deg/s * s = deg

 gyroAngleY = gyroAngleY + GyroY * elapsedTime;

 yaw =  yaw + GyroZ * elapsedTime;
```

The accelerometer of MPU6050 can measure gravitational acceleration along the 3 axes and using trigonometry, we can calculate the angles at which the sensor is positioned.

```
// Calculating Roll and Pitch from the accelerometer data

 accAngleX = (atan(AccY / sqrt(pow(AccX, 2) + pow(AccZ, 2))) * 180 / PI);

 accAngleY = (atan(-1 * AccX / sqrt(pow(AccY, 2) + pow(AccZ, 2))) * 180 / PI);
```

We fuse the accelerometer and the gyroscope data using a complementary filter. Here, we take 96% of the gyroscope data because it is very accurate and doesn't suffer from external forces. The downside of the gyroscope is that it drifts: it introduces errors in the output as time goes on. Therefore, in the long term, we use the data from the accelerometer, 4% in this case, enough to eliminate the gyroscope drift error.

```
// Complementary filter - combine accelerometer and gyro angle values

 roll = 0.96 * gyroAngleX + 0.04 * accAngleX;

 pitch = 0.96 * gyroAngleY + 0.04 * accAngleY;
```

However, as we cannot calculate the yaw value from the accelerometer data, we cannot implement the complementary filter on it, and so the yaw value will drift over time since we cannot use the complementary filter for it.

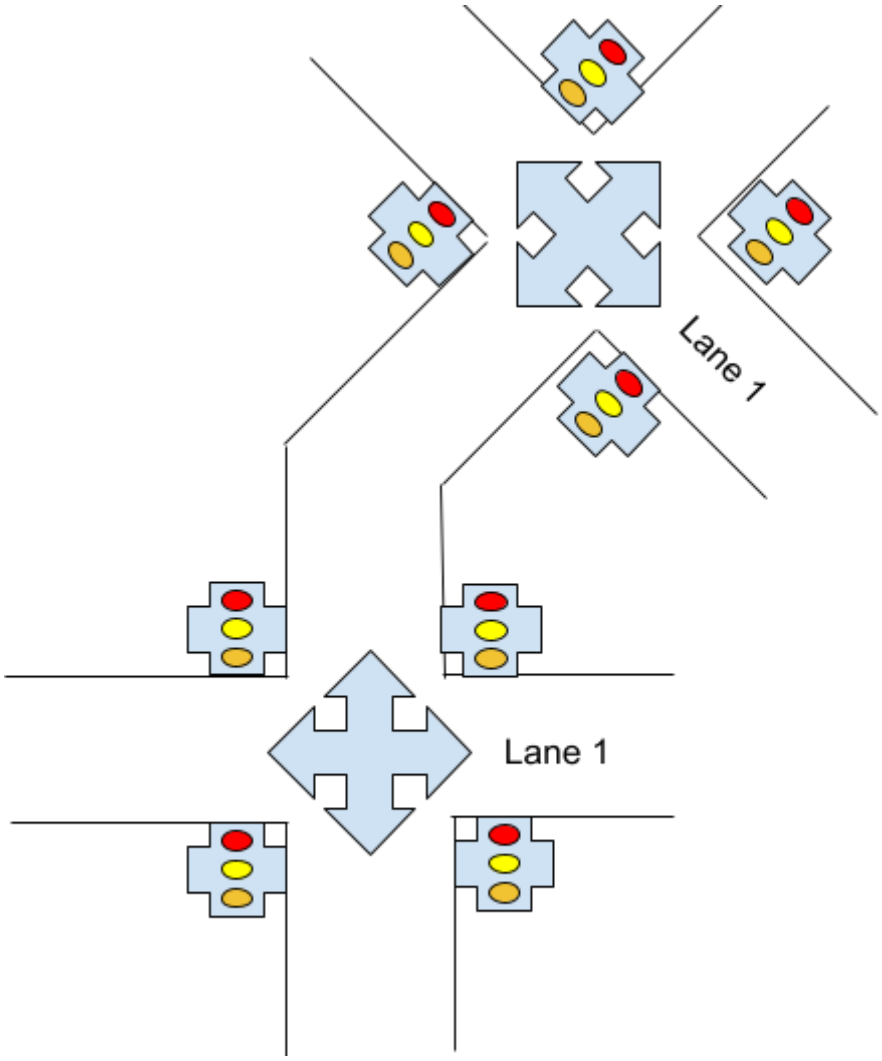
To improve this, we can use an additional sensor, normally a magnetometer which can be used as a long-term correction for the gyroscope yaw drift. However, the MPU6050 actually has a feature that's called Digital Motion Processor (DMP) which is used for onboard calculations of the data and it's capable of eliminating the yaw drift.

The DMP can output quaternions which are used to represent orientations and rotations of objects in three dimensions.

Once we've retrieved data from the DMP, we can use it to get roll-pitch-yaw (RPY) angles. The gravity components need to be first extracted out of the precomputed quaternion data. The gravity and the quaternion values are then passed into the dmpGetYawPitchRoll( ) function to transform them into RPY angles. The output is given in radians so a conversion to degrees can be done if required.

```
mpu.dmpGetQuaternion(&q, fifoBuffer);

mpu.dmpGetGravity(&gravity, &q);

mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

float roll = ypr[0] * 180/M_PI;

float pitch = ypr[1] * 180/M_PI;

float yaw= ypr[2] * 180/M_PI;
```

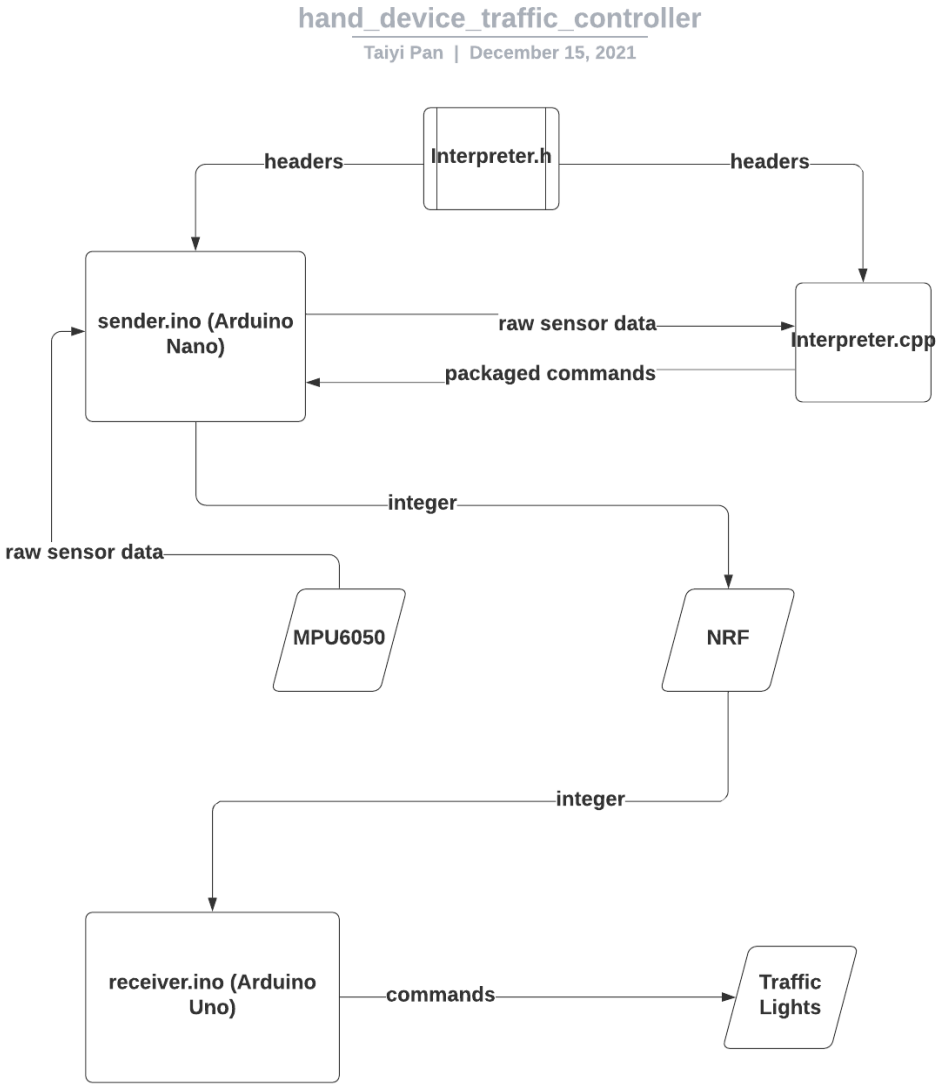
2.7 Calibration Of Lanes



As we can see from the figure above, it shows the problem specific to the alignment of the roads. The figure depicts how the two cross sections are inclined at different angles, therefore the values of one will not work with the other. The way we tackled this problem is by adding a calibration routine. Everytime the user moves to a new cross section, he or she will recalibrate the system and reset the direction of lane 1. The lane 1 is always constant for a given cross section. Once direction for lane 1 is set, the other lanes will take their respective directions in clockwise direction.

2.8 Code Explanation

As shown in the diagram, the code base for the smart gesture based traffic lights controller is highly modularized to maximize code reusability and enhance readability. In terms of Arduino drivers, we have sender.ino and receiver.ino. For mathematical operations, we have the Interpreter class, which is composed of specification file Interpreter.h and implementation file Interpreter.cpp.



For sensors, we have the MPU6050 module. We use the NRF module to facilitate signal transmission. For actuators, we have the traffic lights, which are composed of multitude of LEDs placed in a cross configuration to simulate a real traffic intersection.

The receiver.ino initially starts in automatic mode, so it manages the traffic lights in a regular loop which simulates a normal traffic intersection. On a regular time basis, the intersection either allows vertical traffic or horizontal traffic.

The sender.ino initializes with MPU6050 sensor readings at its zero position. Every 100 milliseconds, sender.ino takes in raw sensor readings and passes the yaw and pitch values to the Interpreter object.

The Interpreter object saves the raw data, then at every moment looks to the past 2 seconds. Lane value is determined by yaw readings. Red and green signals are determined by pitch readings. Red is defined as: for the past 2 seconds, the hand device must be roughly pointing up vertically the whole time. Green is defined as: for the past 2 seconds, the hand device's pitch values must pass pitch up value 3 times, go below minimum threshold 3 times, and go above maximum threshold 3 times. Green signal overrides red signal. Red signal overrides null signal.

Once analysis is complete, the Interpreter object packages the command values into a 2 digit integer for NRF radio transmission. It then

returns the integer command to sender.ino. The sender.ino then sends integer command value to receiver.ino via NRF. Null value is represented by code -1. Code 999 is sent if manual code operation is triggered on sender.ino.

If receiver.ino receives signal code 999 either from sender.ino, or from its own interrupt button, it enters into manual mode, which immediately halts the automatic traffic loop. The mode switch is buffered for 2 seconds to prevent code instability. All the LEDs will pause and stay the same. While in manual mode, all command signals from sender.ino will be interpreted and executed accordingly.

The end result is that while in manual mode, the hand device can control traffic lights remotely via hand gestures and user body rotations. Once manual mode is finished, another control signal 999 can be sent to switch receiver.ino back to automatic mode.

For further details, please see the appendix for the code base (page 24) as well as references for associated Arduino libraries used (page 23).

2.9 Design Critiques

Small Form Factor

A key feature of this design is that it boasts a small form factor that can be easily mounted on gloves, or be worn directly on the hand as shown earlier. The size can be further reduced by using a smaller microcontroller and battery which will further enhance the portability and weight of the transmitting device.

Drift Problem

Even though we utilize the DMP to reduce the drifting problem, it still has some drifting issues.As we don't know how DMP works on the inside, we can only read data from DMP. So to solve this problem, we can either use some filter, such as Kalman filter, or simply add an additional sensor, for instance a magnetometer which can be used as a long-term correction for the gyroscope yaw drift.

Computation Limitations

Arduino Nano, which we used to process raw sensor readings as part of the hand device, is very limited in its computational capacity. Because of that, we can only use hard coded logic to map raw sensor readings into command signals. This approach, while suitable in a controlled environment, can easily break down in the real world since the real world is much more complex with many more variables. No matter how sophisticated the control algorithm is, it

is in the end static, so in the real world, it is not adaptable enough to generalize to all situations.

We propose to use Nvidia Jetson Nano instead of Arduino Nano for the hand device as part of future plans. It is small yet powerful enough to allow us to deploy a dense neural network in the device. The neural net will do the mapping from raw sensor readings into command signals. Since it would be trained upon a large training set based on real world data, it will be much more robust in its mappings, thus allowing the hand device to generalize much better in real world applications.

2.10 Cost Breakdown

Prototype:

Component	Quantity	Cost per unit	Total Cost
Arduino Uno	1	$13.49	$13.49
Arduino Nano	1	$10.99	$10.99
NRF Module	2	$3.95	$7.9
MPU 6050	1	$3.33	$3.33
9V battery	1	$3	$3
Birch wood plank	2	$2	$4
Breadboard	1	$2	$2
Total			**$44.71**
Resistors, LEDs, buttons, wires, and 3D printing materials were sponsored by the Makerspace at NYU Tandon School of Engineering.			

Mass production:

Component	Mass production cost per unit
Arduino Nano	$4.5
Arduino Uno	$4
MPU6050	$0.85
NRF Module	$0.3
Resistors	$1 for 200 pieces
Shipment	$1 per unit
Total cost	**$10.7 per unit**

Mass production components can be obtained from Alibaba as whole sale at the prices mentioned in the table above.

We can move to a custom designed STM chip instead of Arduino Uno to further cut down the cost.

3. CONCLUSION

We fabricated a wearable device that can detect hand gestures, identify the type of hand gesture, and transmit the information to a receiver. The receiver was able to process the received information and control the traffic lights as per the hand gesture.

Any individual wearing the device was able to make a lane's traffic light turn red when performing the "STOP" gesture, and subsequently turn the traffic light green when performing the "GO" gesture.

The device was designed to be easy to use with a small form factor for portability.

4. References

1. Jarzebski. (n.d.). *Jarzebski/Arduino-MPU6050: Mpu6050 Triple Axis Gyroscope & Accelerometer Arduino Library*. GitHub. Retrieved December 15, 2021, from https://github.com/jarzebski/Arduino-MPU6050

2. Jrowberg. (n.d.). *JROWBERG/i2cdevlib: I2C device library collection for AVR/arduino or other c++-based mcus*. GitHub. Retrieved December 15, 2021, from https://github.com/jrowberg/i2cdevlib

3. *NRF24L01 interfacing with Arduino: Wireless Communication*. Arduino Project Hub. (n.d.). Retrieved December 15, 2021, from https://create.arduino.cc/projecthub/muhammad-aqib/nrf24l01-interfacing-with-arduino-wireless-communication-0c13d4

4. New, H. (2021, June 18). *Chapter 4: Traffic control*. New York DMV. Retrieved November 20, 2021, from https://dmv.ny.gov/about-dmv/chapter-4-traffic-control-2

Appendix

```
/*
sender.ino

Sender code for Arduino Nano. Pass raw sensor data from MPU6050 to
Interpreter object, get commands back from Interpreter, then transmit
commands to Receiver via NRF module.
*/

#include "Interpreter.h"
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include <SPI.h>
//#include <String.h>
#include <nRF24L01.h>
#include <RF24.h>

// Arduino Wire library is required if I2Cdev I2CDEV_ARDUINO_WIRE
implementation
// is used in I2Cdev.h
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    #include "Wire.h"
#endif
#include <avr/wdt.h>

// class default I2C address is 0x68
// specific I2C addresses may be passed as a parameter here
// AD0 low = 0x68 (default for SparkFun breakout and InvenSense
evaluation board)
// AD0 high = 0x69
```

```
MPU6050 mpu;
Interpreter interpreter;
const int interruptButton = 4;
const int calibrationButton = 6;
RF24 radio(9, 10); // CE, CSN
int a = -1;
const byte address[6] = "00001";     //Byte of array representing the
address. This is the address where we will send the data. This should
be same on the receiving side.
//MPU6050 mpu(0x69); // <-- use for AD0 high

/*
============================================================================
====
    NOTE: In addition to connection 3.3v, GND, SDA, and SCL, this
sketch
   depends on the MPU-6050's INT pin being connected to the Arduino's
   external interrupt #0 pin. On the Arduino Uno and Mega 2560, this
is
   digital I/O pin 2.
                                                                          *
============================================================================
==== */

#define INTERRUPT_PIN 2  // use pin 2 on Arduino Uno & most boards
#define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
```

```
uint8_t mpuIntStatus;    // holds actual interrupt status byte from MPU
uint8_t devStatus;       // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;     // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorFloat gravity;    // [x, y, z]            gravity vector
float ypr[3];               // [yaw, pitch, roll]    yaw/pitch/roll container and gravity vector



// ================================================================
// ===               INTERRUPT DETECTION ROUTINE                ===
// ================================================================

volatile bool mpuInterrupt = false;       // indicates whether MPU interrupt pin has gone high
void dmpDataReady() {
    mpuInterrupt = true;
}

// ================================================================
// ===               Adding Led control stuff                ===
// ================================================================
```

```
// Pitch, Roll and Yaw values
float pitch = 0;
float roll = 0;
float yaw =0;

// ================================================================
// ===                      INITIAL SETUP                       ===
// ================================================================
void(* resetFunc) (void) = 0; //declare Arduino reset function

void setup() {
    // join I2C bus (I2Cdev library doesn't do this automatically)
    #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
        Wire.begin();
        Wire.setClock(400000); // 400kHz I2C clock. Comment this line if having compilation difficulties(maybe is too fast?,no,it still have to delay)
    #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
        Fastwire::setup(400, true);
    #endif

    // initialize serial communication
    Serial.begin(115200);
       while (!Serial); // wait for Leonardo enumeration, others continue immediately



    // initialize device
      Serial.println(F("Initializing I2C devices..."));//(If you use F() you can move constant strings to the program memory instead of
```

```
the ram)
    mpu.initialize();
    pinMode(INTERRUPT_PIN, INPUT);

    // verify connection
    Serial.println(F("Testing device connections..."));
        Serial.println(mpu.testConnection()  ?  F("MPU6050  connection
successful") : F("MPU6050 connection failed")); //()??

    // load and configure the DMP
    Serial.println(F("Initializing DMP..."));
    devStatus = mpu.dmpInitialize();

     // supply your own gyro offsets here, scaled for min sensitivity
(how to change this?感觉可以改变这里来提高精度？)
    mpu.setXGyroOffset(220);
    mpu.setYGyroOffset(76);
    mpu.setZGyroOffset(-85);
      mpu.setZAccelOffset(1788); // 1688 factory default for my test
chip

    // make sure it worked (returns 0 if so)
    if (devStatus == 0) {
            // Calibration Time: generate offsets and calibrate our
MPU6050
        mpu.CalibrateAccel(6);
        mpu.CalibrateGyro(6);
        mpu.PrintActiveOffsets();
        // turn on the DMP, now that it's ready
        Serial.println(F("Enabling DMP..."));
```

```
        mpu.setDMPEnabled(true);

        // enable Arduino interrupt detection
              Serial.print(F("Enabling interrupt detection (Arduino
external interrupt "));
        Serial.print(digitalPinToInterrupt(INTERRUPT_PIN));
        Serial.println(F(")..."));
                attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN),
dmpDataReady, RISING);
        mpuIntStatus = mpu.getIntStatus();

         // set our DMP Ready flag so the main loop() function knows
it's okay to use it
                  Serial.println(F("DMP ready! Waiting for first
interrupt..."));
        dmpReady = true;

        // get expected DMP packet size for later comparison
        packetSize = mpu.dmpGetFIFOPacketSize();
    } else {
        // ERROR!
        // 1 = initial memory load failed
        // 2 = DMP configuration updates failed
        // (if it's going to break, usually the code will be 1)
        Serial.print(F("DMP Initialization failed (code "));
        Serial.print(devStatus);
        Serial.println(F(")"));
    }

    // configure LED for output
```

```
    pinMode(LED_PIN, OUTPUT);
    wdt_enable(WDTO_2S); // Watch dog 开启看门狗，并设置溢出时间为两秒

    // configure buttons for input
    pinMode(interruptButton, INPUT);
    pinMode(calibrationButton, INPUT);

      radio.begin();                          //Starting the Wireless
communication
     radio.openWritingPipe(address); //Setting the address where we
will send the data
     radio.setPALevel(RF24_PA_MIN);   //You can set it as minimum or
maximum depending on the distance between the transmitter and
receiver.
    radio.stopListening();  //This sets the module as transmitter
}



// ================================================================
// ===                    MAIN PROGRAM LOOP                     ===
// ================================================================

void loop() {
  // if programming failed, don't try to do anything
  if (!dmpReady) return;
  // read a packet from FIFO
   if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) { // Get the Latest
packet
    // display Euler angles in degrees
```

```
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

    yaw = ypr[0] * 180/M_PI;
    pitch = ypr[1] * 180/M_PI;
    roll = ypr[2] * 180/M_PI;
    Serial.print("ypr\t");
    Serial.print(yaw);
    Serial.print("\t");
    Serial.print(pitch);
    Serial.print("\t");
    Serial.println(roll);

    // reset function check
    checkReset();

    // manual mode check
    toggleManualMode();

    //embed interpreter code
    interpretAction();

    // blink LED to indicate activity
    blinkState = !blinkState;
    digitalWrite(LED_PIN, blinkState);
    delay(100);
    wdt_reset(); //喂狗操作，使看门狗定时器复位
  }
}
```

```
//if interruptButton is pressed, send command signal 999 to Receiver
void toggleManualMode() {
  if (digitalRead(interruptButton) == HIGH) {
    //transmit code 999 to receiver Arduino
    int a = 999;
    radio.write(&a, sizeof(a));
    Serial.println(999);
  }
}

//feed yaw and pitch values to Interpreter object, get back lane action commands, then transmit commands to Receiver
void interpretAction() {
  interpreter.feed(yaw, pitch);
  int a = interpreter.getLaneAction();
  //transmit command
  if (a != -1) {
      radio.write(&a, sizeof(a));
      Serial.println(a);
  }
  a = -1; //reset a
}

//if calibrationButton is pressed, call resetFunc to reset Arduino so that MPU6050 makes current yaw the new 0 position
void checkReset() {
  if (digitalRead(calibrationButton) == HIGH) resetFunc();
}
```

```
/*
Interpreter.h

Specification file for Interpreter class.
Interpreter receives sensor data, analyzes it, then returns lane
action commands.
*/

#ifndef INTERPRETER_H
#define INTERPRETER_H

class Interpreter
{
private:
   const static int SIZE = 20; //array size defines attention span;
SIZE * 100ms = time span in ms (20 = 2 seconds)
   const float pitchUp = 85; //pitch value defined as vertical up
position
   const float pitchOffSet = 30; //offset value defines min and max
threshold from vertical up position
  float y; //yaw value, real time
   float p[SIZE]; //pitch array, last 2 seconds, in 20 timesteps,
100ms each timestep
  int lane, action;
  void analyze();
  bool isRed();
  bool isGreen();
  void computeLane();
```

```
  void computeAction();
public:
  Interpreter(); //initialize pinN pinA to -1 null state
  void feed(float, float); //obtain yaw and pitch values from driver
file
  int getLaneAction();
  int getLane(); //return lane designation; return -1 if null
  int getAction(); //return lane action: 0 is red, 1 is green; return
-1 if null
};

#endif
```

```
/*
Interpreter.cpp

Implementation file for Interpreter class.
Interpreter receives sensor data, analyze it, then return lane action
commands.
*/

#include "Interpreter.h"
using namespace std;

//constructor
Interpreter::Interpreter() {
  lane = -1;
  action = -1;
  y = 0;
}

//sensor data input
void Interpreter::feed(float yaw, float pitch) {
  y = yaw;
  for (int i = 0; i < SIZE - 1; i++) {
    p[i] = p[i + 1];
  }
  p[SIZE - 1] = pitch;
  analyze();
}

//analyze data
```

```
void Interpreter::analyze() {
  computeLane();
  computeAction();
}

//use yaw value to determine lane value
void Interpreter::computeLane() {
  //reset lane
  lane = -1;
  //decide lane
  if (y < 45 && y >= -45) lane = 1; //top lane
  else if (y < 135 && y >= 45) lane = 2; //right lane
  else if (y < -135 || y >= 135) lane = 3; //bottom lane
  else if (y < -45 && y >= -135) lane = 4; //left lane
}

//use pitch value to determine action value
void Interpreter::computeAction() {
  //reset pinA
  action = -1;
  //check red and green signals: green override red
  if (isRed()) action = 0;
  if (isGreen()) action = 1;
}

//repackage lane value and action value into a 2 digit integer code
for signal transmission over NRF module
int Interpreter::getLaneAction() {
   if (getLane() != -1 && getAction() != -1) return getLane() * 10 +
getAction();
```

```
  else return -1;
}

//getter
int Interpreter::getLane() {
  return lane;
}

//getter
int Interpreter::getAction() {
  return action;
}

//logic to define red signal: following conditions must be true
//for past 2 seconds, hand device must be roughly pointing up
vertically the whole time
bool Interpreter::isRed() {
  for (int i = 0; i < SIZE; i++) {
    if (p[i] < pitchUp - pitchOffSet || p[i] > pitchUp + pitchOffSet)
      return false;
  }
  return true;
}

//logic to define green signal: following conditions must be true
//for past 2 seconds, hand device's pitch must pass pitchUp 3 times,
go below minimum threshold 3 times, and go above maximum threshold 3
times
bool Interpreter::isGreen() {
  int countlargeP = 0;
```

```
  int countSmallP = 0;
  int countUpP = 0;
  for (int i = 0; i < SIZE; i++) {
    if (p[i] < pitchUp - pitchOffSet)
      countSmallP++;
    if (p[i] > pitchUp + pitchOffSet)
      countlargeP++;
    if (p[i] > pitchUp - pitchOffSet && p[i] < pitchUp + pitchOffSet)
      countUpP++;
  }
  return (countSmallP > 2) && (countlargeP > 2) && (countUpP > 2);
}
```

```
/*
receiver.ino

Receiver code for Arduino Uno. Receives and interprets control signals from Sender. Manages traffic lights between automatic and manual mode.
*/

#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

//declarations
RF24 radio(9, 10); // CE, CSN
const byte address[6] = "00001";
bool manualMode = false;
bool vertical = true;
int n = -1; //integer variable to save data transmission from sender Arduino
int step = 0;
const int maxStep = 30; //value * 100ms traffic orientation cycle
bool bufferStart = false;
int buffer = 0;
const int maxBuffer = 30; //value * 100ms button buffer (during buffer, not responding to further button press; reduce sensitivity; prevent instability)
int yellowTimer = 0;
const int maxYellow = 10;
```

```
//declare button
const int toggleButton = 8;

//global constants
//A0 -> A5 map to pins 14 -> 19
//lane 1 (top)
const int lane1Red = 5;
const int lane1Yellow = 6;
const int lane1Green = 7;

//lane 2 (right)
const int lane2Red = 16; //A2
const int lane2Yellow = 15; //A1
const int lane2Green = 14; //A0

//lane 3 (bottom)
const int lane3Red = 2;
const int lane3Yellow = 3;
const int lane3Green = 4;

//lane 4 (left)
const int lane4Red = 19; //A5
const int lane4Yellow = 18; //A4
const int lane4Green = 17; //A3

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);

  // configure output pins
```

```
  pinMode(lane1Red, OUTPUT);
  pinMode(lane1Yellow, OUTPUT);
  pinMode(lane1Green, OUTPUT);
  pinMode(lane2Red, OUTPUT);
  pinMode(lane2Yellow, OUTPUT);
  pinMode(lane2Green, OUTPUT);
  pinMode(lane3Red, OUTPUT);
  pinMode(lane3Yellow, OUTPUT);
  pinMode(lane3Green, OUTPUT);
  pinMode(lane4Red, OUTPUT);
  pinMode(lane4Yellow, OUTPUT);
  pinMode(lane4Green, OUTPUT);

  //configure input pins
  pinMode(toggleButton, INPUT);

  radio.begin();
   radio.openReadingPipe(0, address);   //Setting the address at which
we will receive the data
   radio.setPALevel(RF24_PA_MIN);       //You can set this as minimum
or maximum depending on the distance between the transmitter and
receiver.
   radio.startListening();                //This sets the module as
receiver
}

void loop() {
  automaticMode();
  checkToggleButton();
  // put your main code here, to run repeatedly:
```

```
  if (radio.available()) { //Looking for the data
    radio.read(&n, sizeof(n)); //Reading the data
    interpret(n);
  }
  bufferIncrement();
  delay(100);
}

//in place of 999 remote signal, create localized 999 signal from local button press
void checkToggleButton() {
  if (digitalRead(toggleButton) == HIGH) interpret(999);
}

//allow button press to buffer so that repeated button press does not destabilize code
void bufferIncrement() {
  if (bufferStart) buffer++; //increment buffer count
//  Serial.println(buffer);
  if (buffer >= maxBuffer) {
    //once buffer reaches threshold, reset buffer
    buffer = 0;
    bufferStart = false;
    }
}

//automatic operating mode
void automaticMode() {
  if (!manualMode) {
    if (vertical) verticalTraffic();
```

```
    else horizontalTraffic();
    step++; //increment step
    if (step >= maxStep) { //switch traffic orientation
      if (vertical) vertical = false;
      else vertical = true;
      step = 0;
      yellowTimer = 0;
    }
  }
}

//lane 1 and lane 3 green, other lanes red
void verticalTraffic() {
  //instant actions
  digitalWrite(lane1Red, LOW);
  digitalWrite(lane1Green, HIGH);
  digitalWrite(lane3Red, LOW);
  digitalWrite(lane3Green, HIGH);
  digitalWrite(lane2Green, LOW);
  digitalWrite(lane4Green, LOW);

  //timed actions
  if (yellowTimer < maxYellow) {
    digitalWrite(lane2Yellow, HIGH);
    digitalWrite(lane4Yellow, HIGH);
  } else {
    digitalWrite(lane2Yellow, LOW);
    digitalWrite(lane4Yellow, LOW);
    digitalWrite(lane2Red, HIGH);
    digitalWrite(lane4Red, HIGH);
```

```
  }
  yellowTimer++;
}

//lane 2 and lane 4 green, other lanes red
void horizontalTraffic() {
  //instant actions
  digitalWrite(lane2Red, LOW);
  digitalWrite(lane2Green, HIGH);
  digitalWrite(lane4Red, LOW);
  digitalWrite(lane4Green, HIGH);
  digitalWrite(lane1Green, LOW);
  digitalWrite(lane3Green, LOW);

  //timed actions
  if (yellowTimer < maxYellow) {
    digitalWrite(lane1Yellow, HIGH);
    digitalWrite(lane3Yellow, HIGH);
  } else {
    digitalWrite(lane1Yellow, LOW);
    digitalWrite(lane3Yellow, LOW);
    digitalWrite(lane1Red, HIGH);
    digitalWrite(lane3Red, HIGH);
  }
  yellowTimer++;
}

//interpret integer transmitted by sender Arduino
void interpret(int n) {
  if (n != -1) {
```

```
    Serial.println(n);
    if (n == 999) {
       if (buffer == 0) { //detect button press with buffer (prevent
instability)
        Serial.println("Mode change detected");
        if (manualMode) manualMode = false;
        else manualMode = true;
        bufferStart = true; //start buffer count
        }
      }
    else {
      if (manualMode) execute(n);
    }
    n = -1; //reset n
  }
}

//execute commands transmitted from sender Arduino
void execute(int command) {
  //break command into lane and action
  int lane = command / 10;
  int action = command % 10;
  //execute accordingly
  switch(lane) {
    case 1: //lane 1 (front 1, then clockwise increment)
      if (action == 0) { //red signal
        digitalWrite(lane1Green, LOW);
        digitalWrite(lane1Yellow, LOW);
        digitalWrite(lane1Red, HIGH);
      } else if (action == 1) { //green signal
```

```
      digitalWrite(lane1Red, LOW);
      digitalWrite(lane1Yellow, LOW);
      digitalWrite(lane1Green, HIGH);
    }
    break;
  case 2: //lane 2
    if (action == 0) { //red signal
      digitalWrite(lane2Green, LOW);
      digitalWrite(lane2Yellow, LOW);
      digitalWrite(lane2Red, HIGH);
    } else if (action == 1) { //green signal
      digitalWrite(lane2Red, LOW);
      digitalWrite(lane2Yellow, LOW);
      digitalWrite(lane2Green, HIGH);
    }
    break;
  case 3: //lane 3
    if (action == 0) { //red signal
      digitalWrite(lane3Green, LOW);
      digitalWrite(lane3Yellow, LOW);
      digitalWrite(lane3Red, HIGH);
    } else if (action == 1) { //green signal
      digitalWrite(lane3Red, LOW);
      digitalWrite(lane3Yellow, LOW);
      digitalWrite(lane3Green, HIGH);
    }
    break;
  case 4: //lane 4
    if (action == 0) { //red signal
      digitalWrite(lane4Green, LOW);
```

```
        digitalWrite(lane4Yellow, LOW);
        digitalWrite(lane4Red, HIGH);
      } else if (action == 1) { //green signal
        digitalWrite(lane4Red, LOW);
        digitalWrite(lane4Yellow, LOW);
        digitalWrite(lane4Green, HIGH);
      }
      break;
  }
}
```