WeMove

Head Gesture Controlled Wheelchair

CS4EP, Project Report

Vincenzo Ciavolino, Niki Di Giano, Eleonora Erriquez

A.Y. 2019/2020

Contents

1 The Problem Statement

1.1 Introduction

Providing those affected by motor impairment with sufficient means to live is a central problem in medicine, automation, and biomedical engineering. Over the years, a number of solutions have been provided: for disabilities that only affect the lower part of the body, motorized wheelchairs with control pads were invented and distributed on the market in various forms.
In this paper we describe the approach we took for the problem of providing the same amount of control over the wheelchair for those who have lost the ability to move any limb, and whose movement is restricted to the neck and head area. From our work was born WeMove.

1.2 Primary Assumptions

In the development of our proof of concept, we have assumed that the user has total freedom of motion for the neck and head, while it is not necessary for the rest of the body to move. This means that the only motion we need to retrieve with sensors is the one of the head.
A motor impaired individual cannot be self-sufficient enough to live independently. We know that. This means that in any case, it is necessary for the individual to have at least one caretaker with them at all times. This caretaker is assumed to be the one to provide the user with the necessary assistance to wear the hat included in the project, as well as offering supervision in situations where the user can endanger themselves.
This project cannot liberate a motor impaired patient from their impairment, but it can offer a degree of independence that exists as long as there is life. And while there is life, there's hope.

1.3 The Components

Here is a list of the components that have been used to put together this project.

Smart-Car:
1 x WeMos D1 Mini;
2 x 18650 Li-Ion 4.2V 8800mAh Batteries;
1 x L293D Quadruple Half-H Driver Chip;
2 x Geared Motors 3-12V 60Rpm ;
1 x Toggle Switch;

Smart-Hat:
1 x WeMos D1 Mini;
1 x LIR2023 Li-Ion 3.6V 50mAh Battery;
1 x MPU6050 Six-Axis Accelerometer + Gyroscope Sensor;
1 x TP4056 Li-Ion Battery Charger;
1 x Toggle Switch;

2 The Realization

2.1 The Motion Detection System

Our primary focus was on the realization of a proper sensor to identify the motion of the neck and head. Initially, we proposed an accelerometer placed on a collar to be put around the user's neck, however we found a number of problems with this approach. Primarily, the rotation of the neck around the z axis (yaw angle) is a natural motion, which can be caused even by mere interest of the user in what's happening on its left and right. Using such motion to give commands to the wheelchair meant that the user was restricted in performing this natural motion unless they wanted to impart precisely that command, which is not ideal. Moreover, while the neck rotates and twists, the changes in angle were too small for our chosen accelerometer (MPU6050) to be stable.
This lead us to find a different solution, one that would pick up on the motion of the head rather than the neck. We immediately thought of embedding the sensor in a hat, and use the rotation of the head along the x and y (roll and pitch angles) to guide the wheelchair: in particular, bending the head left or right would rotate the chair, and bringing the head forwards causes the chair to accelerate.

Figure 1: The motion sensor glued on the hat. (On the right the system under charging process)

These motions are not unnatural enough to prevent a motion-impaired user from performing them, but they are not natural enough to be performed by mistake. In particular, we reserved no actions for an upwards motion of the head since this is a necessary movement in order to interact with other people or inspect the environment, assuming the average

person to be taller than the user sitting in the wheelchair.
Implementing the commands meant that we needed to specify a range of angles where the rotation of the head would be recognized. This immediately brought us to a problem: if the user's hat is not placed perfectly on the head, then the angles read by the accelerometer while the head is upright are nonzero; this means that for a non-perfectly aligned sensor, for example if the sensor leans to the left, it takes more effort to issue a command by rotating the head towards the right than towards the left.
For this reason, during the first twenty seconds in which the sensor stabilizes, we decided to implement a simple de-bias algorithm that takes an average value of the head's rotation at rest and sets it as the zero point.
We designed the system to be fully autonomous, so a microcontroller with Wi-Fi connectivity has been chosen and battery power has been included.
The battery is a coin cell of 3.6V nominal voltage, is based on Li-Ion and is rechargeable. To charge it a TP4056 charger board has been introduced, but since this device is designed to charge big Li-Ion batteries with a 1A current, which is too much for our tiny battery, a modification has been performed. The $1.3k\Omega$ resistor that sets the charging current has been desoldered and replaced with a $100k\Omega$ one, so the charging current is reduced to 15mA, an appropriate value according to the battery specifications.

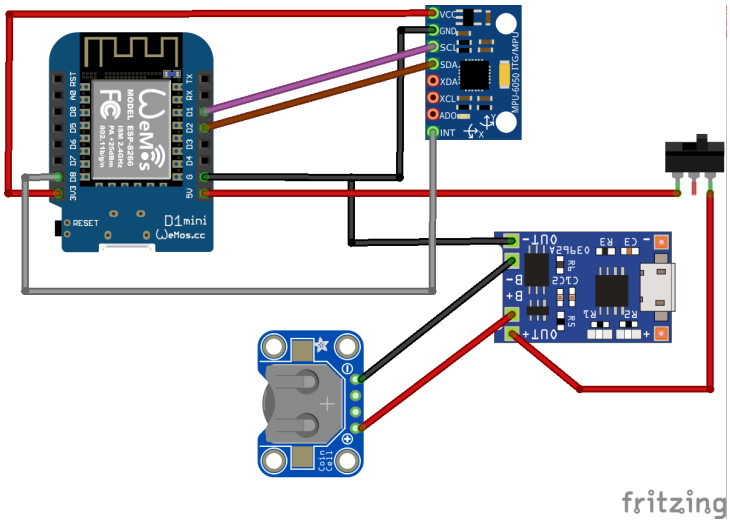

Figure 2: The Fritzing scheme for the motion sensor on the hat.

2.2 The Wheelchair

For what concerns the wheelchair, we thought to build a simple model of a car made of two front wheels joined by a fixed shaft and two independent rear wheels driven by geared motors.
The motors are controlled by a L293D chip, which contains four half-H bridges. This device is made of Darlington transistors and is designed to drive a wide array of inductive loads such as relays, solenoids, DC and bipolar motors. The package has four ground

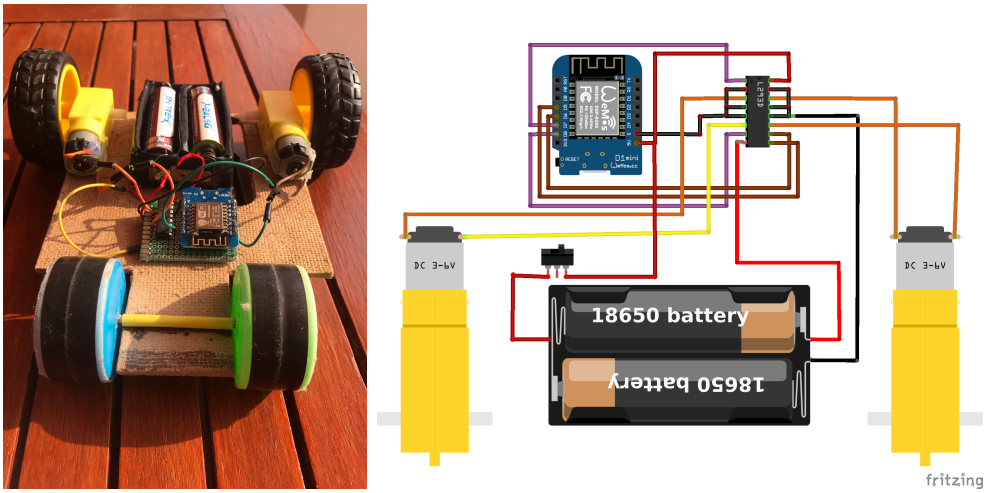

Figure 3: The model for the remote-controlled car and the Fritzing scheme.

pins, that we connected to the common ground of the microcontroller and batteries; two output pins for each motor, that can deliver high currents; two input pins for each motor, controlled digitally by the microcontroller; two enable pins that allow to provide PWM signals to the motors, so that acceleration can be implemented; a 5V input pin to supply power to the internal logic and a 4.5V to 36V input pin to power the motors.
The microcontroller is a WeMos D1 Mini and is equipped with Wi-Fi connectivity, it receives data from the sensor and based on these information it produces the proper control outputs directed to the L293D.
The prototype is powered by two 18650 Li-Ion batteries of 4.2V nominal voltage, that have been chosen for their capability to deliver high currents and for the possibility to be re-charged. One of those cells powers the microcontroller and the driver chip, while the series of the two provides the motors of 8.4V at full charge.

2.3 The Connection

After realizing and assembling the two components of our device, it was time to let them talk to one another. This immediately presented itself as one of the most important choices: wired or wireless?
We analyzed both solutions before taking our decision. While a wired connection would provide a much simpler final product, lower production costs and faster response, there is a huge drawback that we could not ignore: a wire leading from the head to the wheelchair can be pulled, it can twist, and it can otherwise cause the attached sensor to experience forces that should not be there. The continued motion of the head, if the wire is not properly sized, can cause the sensor to be displaced and even pull on the hat with enough of a force to take it off.
A wireless connection, on the other hand, would have none of these problems. The cost and the power dissipation are higher than the previous case, however the gain in stability is critical and makes up for this increased cost. We also have a marginal gain in the

aesthetics of the whole.
For these reasons we have decided to adopt the wireless connection over the wired one. This didn't come without its own set of choices, in particular for the choice of the wireless devices to use. We adopted the WeMos D1 Mini chip, which is a microcontroller based on ESP8266, a great platform for IoT applications. The selected board is small enough to be very cheap and compact, but has the necessary resources in terms of number of GPIOs, Flash memory size and Wi-Fi connectivity needed for the project. We installed one of these boards either on the hat and the car model. The first was configured as an access point that creates a Wi-Fi network with its SSID and password, the second as station that automatically connects to the access point. To make them talk to each other we chose the UDP transport protocol, so setting the IP addresses and the local port the unidirectional connection from the board on the hat to the one on the car has been established. UDP is an internet protocol built for speed rather than reliability, so we adopted it because of the time-sensitive nature of our problem and because dropping packets was preferable to waiting for packets delayed due to retransmission; even in the event of the loss of a packet, the performance of the device as a whole is not impacted.

3 The Code

3.1 Smart_hat.ino

```
// Libraries importing
#include <Arduino.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include <MPU6050.h>
#include "Wire.h"
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

WiFiUDP Udp;
MPU6050 mpu;

// MPU6050 processor variables
bool dmpReady = false; // set true if DMP (Digital Motion Processor)
    ↪ 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

int i = 0;

// Orientation/motion vars
Quaternion q; // [w, x, y, z] quaternion container
VectorInt16 aa; // [x, y, z] accel sensor measurements
VectorInt16 aaReal; // [x, y, z] gravity-free accel sensor measurements
VectorInt16 aaWorld; // [x, y, z] world-frame accel sensor measurements
VectorFloat gravity; // [x, y, z] gravity vector
float euler[3]; // [psi, theta, phi] Euler angle container
float ypr[3]; // [yaw, pitch, roll] yaw/pitch/roll container and gravity
    ↪  vector
volatile bool mpuInterrupt = false; // indicates whether MPU interrupt
    ↪ pin has gone high

// OUR VARIABLES
float angles_sum[3] = {0.0, 0.0, 0.0}; // [x, y, z]

//WI-FI network login credentials
```

```
const char *ssid = "CAR_AP";
const char *password = "SmartCar";
unsigned int localPort = 2390;

void setup() {
  Wire.begin(); //I2C connection initialization
  Serial.begin(9600); //Serial connection initialization

  // MPU6050 Gyroscope-Accelerometer sensor initialization
  while (!Serial); // wait for Leonardo enumeration, others continue
     ↪ immediately
  Serial.println(F("Initializing␣I2C␣devices..."));
  mpu.initialize();
  Serial.println(F("Testing␣device␣connections..."));
  Serial.println(mpu.testConnection() ? F("MPU6050␣connection␣successful"
     ↪ ) : F("MPU6050␣connection␣failed"));
  Serial.println(F("Initializing␣DMP..."));

  devStatus = mpu.dmpInitialize();
  mpu.setXGyroOffset(220);
  mpu.setYGyroOffset(76);
  mpu.setZGyroOffset(-85);
  mpu.setZAccelOffset(1788); // 1788 factory default for my test chip

  if (devStatus == 0) {
    // turn on the DMP, now that it's ready
    Serial.println(F("Enabling␣DMP..."));
    mpu.setDMPEnabled(true);
    Serial.println(F("Enabling␣interrupt␣detection␣(Arduino␣external␣
       ↪ interrupt␣0)..."));
    attachInterrupt(digitalPinToInterrupt(15), dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();
    Serial.println(F("DMP␣ready!␣Waiting␣for␣first␣interrupt..."));
    dmpReady = true;
    packetSize = mpu.dmpGetFIFOPacketSize();
  } else {
    Serial.print(F("DMP␣Initialization␣failed␣(code␣"));
    Serial.print(devStatus);
    Serial.println(F(")"));
  }

  // UDP connection initialization
  WiFi.mode(WIFI_AP);
  WiFi.softAP(ssid, password);
  Udp.begin(localPort);
```

```
}

void loop() {

  // Sensor data reading
  if (!dmpReady) return;
  mpuInterrupt = false;
  mpuIntStatus = mpu.getIntStatus();
  fifoCount = mpu.getFIFOCount();
  // Serial.println(fifoCount);
  if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
    mpu.resetFIFO();
    Serial.println(F("FIFO␣overflow!"));
  } else if (mpuIntStatus & 0x02) {
    while (fifoCount < packetSize) {
      fifoCount = mpu.getFIFOCount();
    }
    mpu.getFIFOBytes(fifoBuffer, packetSize);
    fifoCount -= packetSize;
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

    int x = 0;
    int y = 0;
    int z = 0;

    // Initial calibration
    if (!(i < 1999)) {
      x = ypr[0] * 180 / M_PI - angles_sum[0];
      y = ypr[1] * 180 / M_PI - angles_sum[1];
      z = ypr[2] * 180 / M_PI - angles_sum[2];
    } else {
      x = ypr[0] * 180 / M_PI;
      y = ypr[1] * 180 / M_PI;
      z = ypr[2] * 180 / M_PI;
    }

    // Values print for debug
   // Serial.print(x); Serial.print("/"); Serial.print(y); Serial.print
      ↪ ("/"); Serial.println(z); Serial.print("\n"); Serial.println(i)
      ↪ ;

    // UDP packet creation and transmission
    Udp.beginPacket("192.168.4.2", localPort);
```

```
    Udp.print(x); Udp.print("/"); Udp.print(y); Udp.print("/"); Udp.print(
        ↪ z);
    Udp.endPacket();

  }
  // Initial mean values computation
  if (i < 2000) {
    if (i >= 1990) {
      angles_sum[0] += ypr[0] * 18 / M_PI;
      angles_sum[1] += ypr[1] * 18 / M_PI;
      angles_sum[2] += ypr[2] * 18 / M_PI;
    }
    i++;
  }
  mpu.resetFIFO(); //FIFO buffer reset to avoid overflow
}


// Interrupt function
ICACHE_RAM_ATTR void dmpDataReady()
{
  mpuInterrupt = true;
}
```

3.1.1 Declared variables

Udp and *mpu* are classes provided by the respective libraries (*WiFiUdp.h* and *MPU6050.h*) and store the methods used for the initialization and manipulation of data throughout the algorithm. They require a number of variables to be defined (*dmpReady* through *localPort*). Every variable's function is commented in the code.

3.1.2 Setup()

In Setup, all the relevant quantities are initialized by calling the relative functions provided by the I2C and MPU dedicated libraries. We also initialize the WiFi connection with the specified login credentials and the UDP connection.

3.1.3 Loop()

The Loop function is tasked with (fifo) Then, the buffer is used by the DMP (Digital Motion Processor) inside the sensor chip to obtain and store a quaternion vector, stored

in the variable *q*, and the gravity vector *gravity*. Those values are employed by the processor to compute the yaw/pitch/roll rotation angles which are then stored in the *ypr* vector. The stabilization or calibration process is performed with the data obtained from the last 10 cycles of the first 2000 cycles of the algorithm; we have decided to use these values because they provide the best results with the minimal memory usage. If we haven't reached 2000 cycles yet, we need to store the current, non-stabilized values of the yaw/pitch/roll vector in order to feed them into the next section, which is precisely the section that performs the average over said 10 cycles. If we have reached the 2000th cycle, we take the averaged yaw/pitch/roll vector and subtract it from the current one.

3.1.4 dmpDataReady()

This function is a necessary definition since it is to be fed into the function *attachInterrupt* which binds the interrupt pin to the interrupt event; if there's an interrupt, the pin is set high.

3.2 Smart_car.ino

```
// Libraries importing
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

// Motor pins declaration
int motor_left = 15;
int motor_right = 12;

const int enableleft = 13;
const int enableright = 14;

// Motor control variables
const int rollTreshold_R = 20;
const int rollTreshold_L = -20;
const int pitchTreshold = -10;
bool calibrationEnd = false;

// WI-FI network login credentials
#ifndef STASSID
#define STASSID "CAR_AP"
#define STAPSK "SmartCar"
#endif

unsigned int localPort = 2390; // local port to listen on
```

```
// buffers for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE + 1]; //buffer to hold incoming
    ↪ packet,

int x, y, z;

WiFiUDP Udp;

void setup() {

  // Serial connection setup
  Serial.begin(9600);

  // WI-FI connection setup
  WiFi.mode(WIFI_STA);
  WiFi.begin(STASSID, STAPSK);
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(500);
  }
  Serial.print("Connected! IP address: ");
  Serial.println(WiFi.localIP());

  // UDP connection setup
  Serial.printf("UDP server on port %d\n", localPort);
  Udp.begin(localPort);

  // Motor control pins initialization function call
  initializeMotorPins();

}

void loop() {

  // if there's data available, read a packet
    int packetSize = Udp.parsePacket();
    if (packetSize) {

  // read the packet into packetBufffer
    int n = Udp.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
    packetBuffer[n] = 0;

  // Print on serial monitor contents of the received packet
    Serial.println(packetBuffer);
```

```
  }

  // Packet values char-to-int conversion
    char * token = strtok(packetBuffer,"/");
    int i=0;
    int xyz[3];
    while( token != 0 ) {
        xyz[i]= atoi(token);
        i++;
        token = strtok(0, "/");
   }
     x = xyz[0];
     y = xyz[1];
     z = xyz[2];
     int vel = map(y, 0, 32, 0, 1023);

  // Motor control function call
    if ((x==0 && y==0 && z==0)||(calibrationEnd == true)) {
    calibrationEnd = true;
    digitalWrite(2, LOW);
    motorControl(x, y, z, vel);
  }
  else if (calibrationEnd == false){
    blink_LED();
    }
}


// Motor control pins initialization function
  void initializeMotorPins()
  {
    pinMode(motor_left, OUTPUT);
    pinMode(motor_right, OUTPUT);
    pinMode(enableleft, OUTPUT);
    pinMode(enableright, OUTPUT);
    pinMode(2, OUTPUT);

  }

//Motor control function
void motorControl(int x, int y, int z, int vel)
{
  if (z < rollTreshold_R && z > rollTreshold_L && y > pitchTreshold){
      motor_stop();
    }
```

```
  else if (y > pitchTreshold && (z > rollTreshold_R)) {
      turn_right();
  }
  else if (y > pitchTreshold && (z < rollTreshold_L)) {
      turn_left();
  }
  else if (z < rollTreshold_R && z > rollTreshold_L && y < pitchTreshold)
     ↪  {
      drive_forward(vel);
    }


}

void motor_stop(){
analogWrite(enableleft, 0);
analogWrite(enableright, 0);

digitalWrite(motor_left, LOW);
digitalWrite(motor_right, LOW);
    Serial.print("Motor stop\n");

}

void drive_forward(int vel){
  analogWrite(enableleft, -vel);
  analogWrite(enableright, -vel);

  digitalWrite(motor_left, HIGH);
  digitalWrite(motor_right, HIGH);
      Serial.print("Drive forward\n");

}

void turn_left(){
  analogWrite(enableleft, 1023);
  analogWrite(enableright, 1023);

  digitalWrite(motor_left, LOW);
  digitalWrite(motor_right, HIGH);
      Serial.print("Turn left\n");

}

void turn_right(){
```

```
  analogWrite(enableleft, 1023);
  analogWrite(enableright, 1023);

  digitalWrite(motor_left, HIGH);
  digitalWrite(motor_right, LOW);
      Serial.print("Turn␣right\n");

}

void blink_LED(){
  digitalWrite(2, HIGH);
  delay(50);
  digitalWrite(2, LOW);
  delay(50);
  }
```

3.2.1 Declared variables

As in the first script, we define the *Udp* object and the quantities used by the motor functions. *packetBuffer* stores the incoming data sent over UDP communication.

3.2.2 Setup()

In Setup we initialize the WiFi connection using the SSID and password (which must match the ones sent over by the smart hat), the UDP connection and the motor pins, with the *initializeMotorPins* function.

3.2.3 Loop()

In Loop, we first gather the data from the UDP packet, split it into three pieces, which correspond to the yaw/pitch/roll values sent over by the hat, and turn them into usable integers. Then, we call the *motorControl* function with the obtained integers, but only if the calibration step is completed. This step is considered complete when the values of the received angles are perfectly null.

3.2.4 initializeMotorPins()

This function is a container for the initialization relative to the motor pins.

3.2.5 motorControl()

This function sets up the thresholds for the activation of the motion functions (detailed below). The thresholds defined at the beginning of the algorithm provide the boundaries in which the relative motion is performed.

3.2.6 Motion functions

The motion unctions (*turn_left, turn_right, motor_stop, drive_forward*) control the voltage of the motor pins in order to manipulate the motor's rotation. The meaning of each function is self-explanatory.

3.2.7 blink_LED()

A simple function that makes the internal LED of the board, which is connected to pin 2, blink. It's used to visually inform that the connection has been established and stabilization process is being performed.

4 Future Developments

In this project, we tried to keep it simple and deliver a product that has the necessary functionality of any fully developed prototype. However, there are many possibilities to expand on the original idea.
For starters, we can expand the stabilization process to include not only the initial average, but also subsequent averages at regular intervals. In fact, with the motion of the head, the hat is bound to move away a little from the initial position and therefore, the user experience can change over time. This however is no easy task, as during motion the head may move around in a chaotic pattern, therefore careful data manipulation is required to extract the "resting position" from this motion.
Another possible development is to expand the category of body parts that can be used with the algorithm: gloves can capture the motion of the hands, anklets can capture the motion of the feet, and other specialized sensors. These would require the adaptation of the threshold values for the motor control, as well as a better stabilization algorithm that doesn't rely on averaging. In fact, the current averaging algorithm only works for an orientation of the sensor similar to the vector of gravity; the reason being that fundamentally, the function that should be used is much more complicated and a "linear" algorithm only works as a Taylor expansion to the first order.
A further upgrade would be to provide the system with more sensors, e.g. proximity sensors (PIR, ultrasound, lidar, etc..) to avoid the case of collision with walls or big objects; a digital camera to allow monitoring of the user by caretakers or parents; for a much more advanced implementation the camera and the sensors could also give the input data for artificial intelligence algorithms that can add self-driving capabilities to the system.
Finally, gesture detection algorithms could be introduced in order to encode certain movements of the sensor to certain pattern of motion of the wheelchair, like complex manoeuvres or fixed trip routes.
Currently, the device is integrated into a remote-controlled car. However, it would be possible to modify the product slightly and install the motors on a non-smart wheelchair. Wheelchairs also happen to be expensive if compared to the cost of the current prototype. Therefore, installing on a pre-existing chair would keep the production cost to a minimum and therefore, from a marketing point of view, allow for a much broader consumer market.

5 Conclusions

This project allowed our group to join forces and think critically to come up with a reachable solution to an important problem. We have faced many problems during the development of the prototype, many of which we have solved using the background we have built upon over the years.
Our prototype works as intended and has overcome the critical steps in the realization, namely the stabilization issue, the problem of establishing a connection between two microcontrollers and the proper driving of motors. The project lended itself to very clear places where it could be expanded upon without having to get rid of any of the original ideas, and while we couldn't pursue all those possibilities we are very happy with our proof of concept.