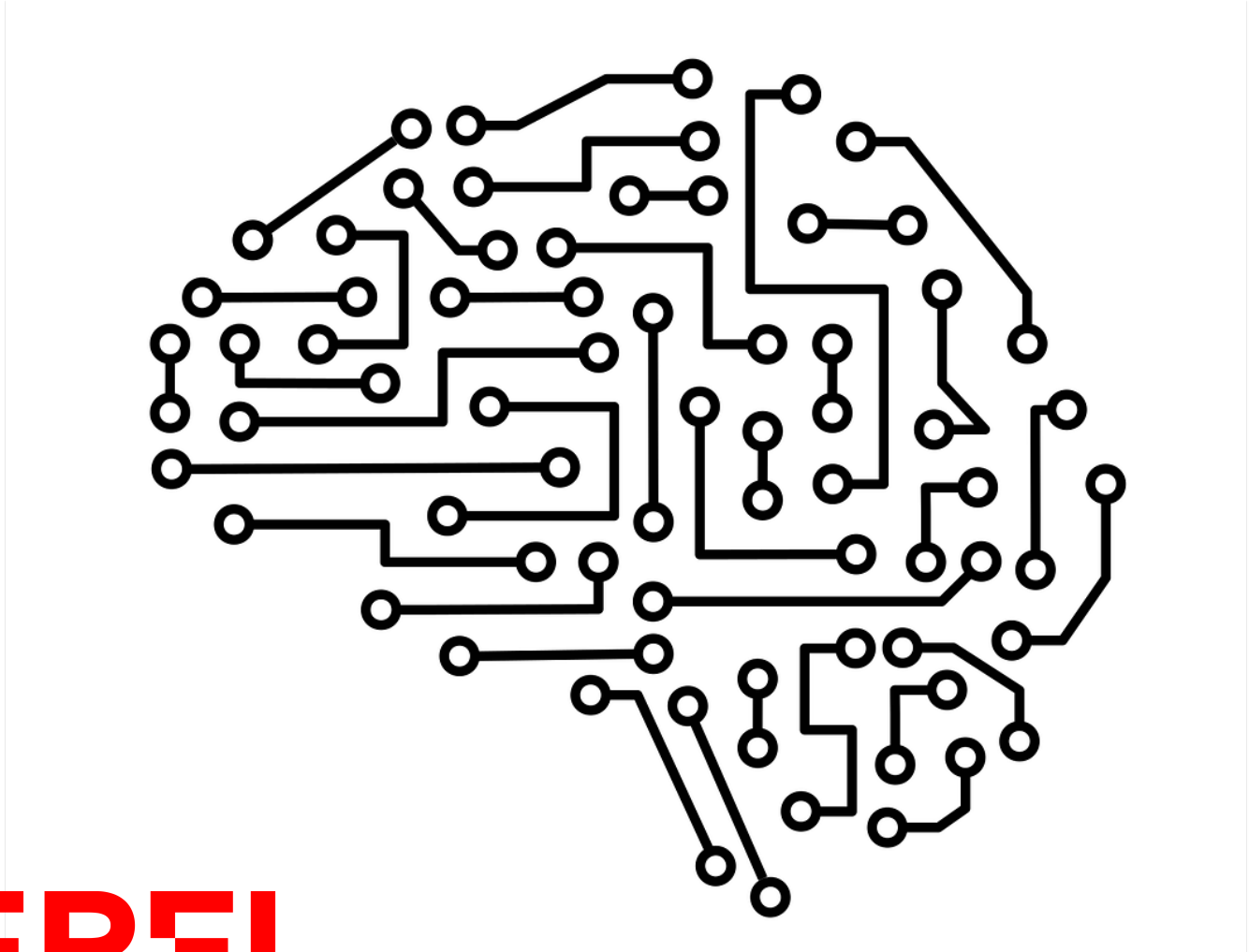
Mind-Controlled Robot

Engineering and Interfacing
with the Brain

CS-358: Making Intelligent Things

Michael Freeman
Emile Janho Dit Hreich
Eric Saikali
Marin Vogelsang
Maxime Zammit



EPFL
Ecole Polytechnique Fédérale de Lausanne

Mind-Controlled Robot

Engineering and Interfacing with the Brain

by

Michael Freeman
Emile Janho Dit Hreich
Eric Saikali
Marin Vogelsang
Maxime Zammit

Student Name	Student Number
Michael Freeman	313215
Emile Janho Dit Hreich	309994
Eric Saikali	326450
Marin Vogelsang	281885
Maxime Zammit	310251

Instructor:	C. Koch
Teaching Assistant:	L. Klein, F. Stella
Project Duration:	March, 2022 - June, 2022
Faculty:	School of Computer and Communication Sciences, EPFL

EPFL

Preface

The quest of understanding how the human mind and brain work has gained much attention over the last decades, increasingly so also from computer scientists, physicists, and engineers. Reading out and interfacing with the brain is an interdisciplinary challenge that can lead to the expansion of our understanding of ourselves and others. From the practical perspective, it also promises to lead to the emergence of tangible applications that could improve our lives in the future. This project aims at providing a medium in which technologies to interface with the brain are made available to all makers, to enable them to embark on this quest even if they do not have access to any fully-equipped professional laboratories.

Michael Freeman
Emile Janho Dit Hreich
Eric Saikali
Marin Vogelsang
Maxime Zammit
EPFL, July 2022

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
EEG	Electroencephalography
EMG	Electromyography
IMU	Intertial Measurment Unit
BLE	Bluetooth Low-Energy
SVM	Support Vector Machine
ROS	Robot Operating System
I2C	Inter-Integrated Circuit, (eye-squared-C)

1

Project Goal

The final goal of this project is to be able to control a LEGO vehicle through signals obtained from the brain as well as through recordings of muscle activity and head movements. More specifically, as part of our final product, we wish to enable a user to control the speed of a car by increasing or decreasing his/her mental focus, as picked up by electroencephalography (EEG) signals; the direction of the car by rotating his/her head to the left or the right, as picked up by the inertial measurement unit (IMU); and switch the car from moving forward to moving backward, and vice versa, by eliciting a strong arm movement, as picked up by the electromyography (EMG) signals and inspired by the movement car drivers make when switching gears in a non-automatic car. This scheme is illustrated in Figure 1.1

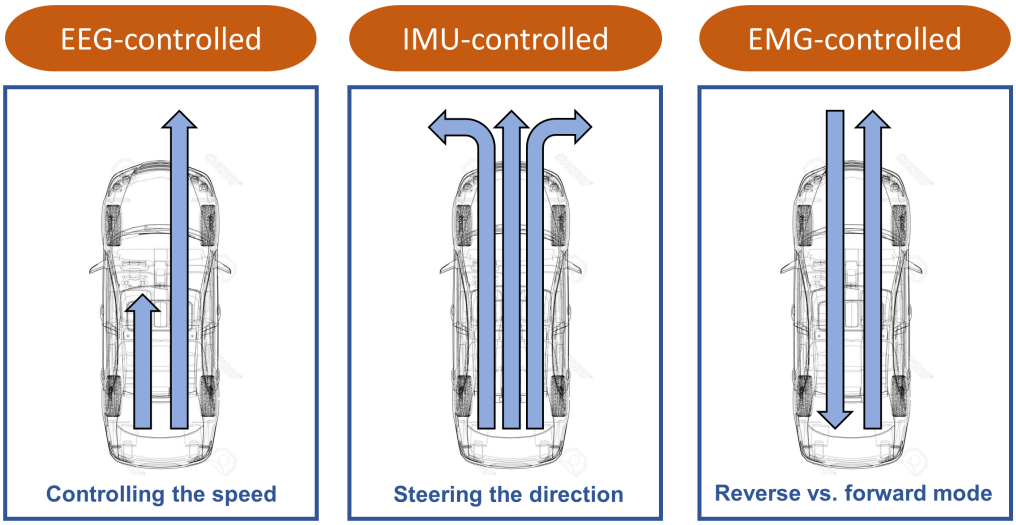


Figure 1.1: Illustration of car control: the speed of the car is controlled by EEG signals (the higher the mental focus, the faster the car moves), the left/right direction by IMU signals (switching from moving straight to moving left or right when a strong directional tilt is detected in the IMU), and the forward/backward direction by EMG signals (switching between the two modes when a strong EMG signal is detected). The sketch of the car is taken from 123rf.com.

2

Overview

This project is designed to be accessible to all makers with basic knowledge of electronics and relatively advanced programming skills who are eager to discover the field of neuroscience and interfacing with the brain. In this section, we provide a brief overview of the types of material and skills that are required to complete the different parts of the project.

Section 3: Signal Acquisition and Processing

As motivated in the Introduction, we seek to control a LEGO vehicle on the basis of recorded brain activity, muscle activity, as well as signals derived from a head-mounted gyroscope. The [Crowd Supply WallySci E3K Bio-Sensing Platform] appeared to be the best solution, incorporating all of the above while running at a cost of only 154 CHF and offering excellent flexibility with regard to coding. In addition, we ordered [conductive gel] to use pad replacements and, considering that the WallySci E3K Bio-Sensing Platform is shipped with only a single set of electrodes, we purchased [additional ones].

Section 4: Car Control and Communication

We used an Arduino UNO board for the car control, along with a Bluetooth module (HC-05) for communicating with a backend computer, which receives the input from the headset via Bluetooth as well.

Section 5: DIY EEG Circuit

In addition to using the WallySci E3K Bio-Sensing Platform for recording EEG data, we also built our own EMG-ECG circuits, as inspired by the low-cost do-it-yourself guides [1] [2]. The detailed material list can be found in Section 5.

Section 6: Headset

In order to mount the gyroscope sensor, the EEG sensor, and the board to the head, while providing comfort, we designed and 3D-printed an EEG headset. We used Fusion360 to make the design, then we generated the GCode files with PrusaSlicer and printed it with a Prusa Mark 3 3D Printer. We used PETG and PLA materials. In addition, we just used some hot glue and m3 screws to assemble everything.

Section 7: Car Design

As the basis for our car, we used the car from the individual project from this course, and made several changes to it: we used a gear motor, which improved the mobility of the car, and we added a battery box with 6 1.5V AA batteries each.

3

Signal Acquisition and Processing

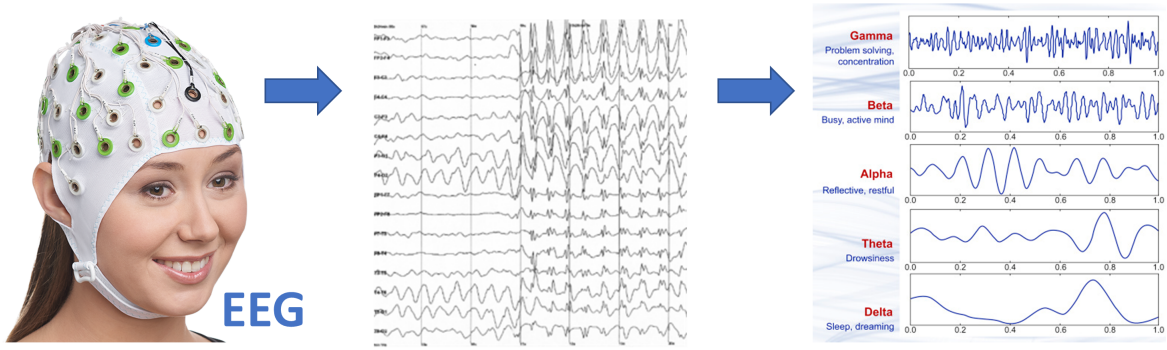
Electroencephalography

Background

Electroencephalography (EEG) [3] allows for the recording of electrical activity on the scalp, which is representative of underlying brain activity. This electrical activity, as typically recorded with non-invasive electrodes placed within an EEG cap, can be analyzed, for instance, with regard to the distribution of frequencies making up the electrical signal. These frequencies provide a strong signature that can be, and has been, linked to different levels of cognitive alertness: whereas very high frequencies, such as those present in beta or gamma waves, have been associated with mental focus and problem solving, low-frequency theta waves are associated with drowsiness, and very low frequencies in the delta range with deep sleep (see, e.g., [4] or [5] for reference; see Figure 3.1.(a) for illustration). This renders EEG recordings and subsequent frequency analyses interesting diagnostic tools but also allows for some recreational derivations: in this project, to use the extracted frequency response to control the speed of our LEGO vehicle.

Setup used

EEG systems used in neuroscientific laboratories are typically equipped with many (often 64 or 128) electrodes (see, e.g., [6]; see Figure 3.1.(a) for visualization). Having many electrodes spread across the scalp is crucial when examining the finer spatial dynamics of neural signals, for instance, when attempting to reconstruct where in the brain a given signal had originated from. However, as we only seek to examine the overall frequency response, we utilized the relatively low-cost Crowd Supply WallySci E3K Bio-Sensing Platform, whose EEG module comes with two electrodes to be applied to the forehead as well as a reference electrode to be applied behind the ear. Several factors encouraged us to attempt to measure frequency-based cognitive alertness using this relatively simple setup. First, there do exist commercial games, such as Mindflex [7] (see Figure 3.1.(b)), that, too, can be 'mind-controlled' with a mobile EEG headset. Second, the do-it-yourself guides [1], [2] (see Figure 3.1.(c)) suggested the feasibility of deriving meaningful signals from even more minimalistic and lower-cost portable electrode systems. Finally, while, as stated above, EEG caps with many electrodes tend to be the standard in many neuroscientific laboratories, it is worth pointing out that clinical sleep laboratories, in which the overall frequency response is of great interest to judge, for instance, sleep stages, often only apply relatively few electrodes as well (see, e.g., [8], [9], and Figure 3.1.(d)).



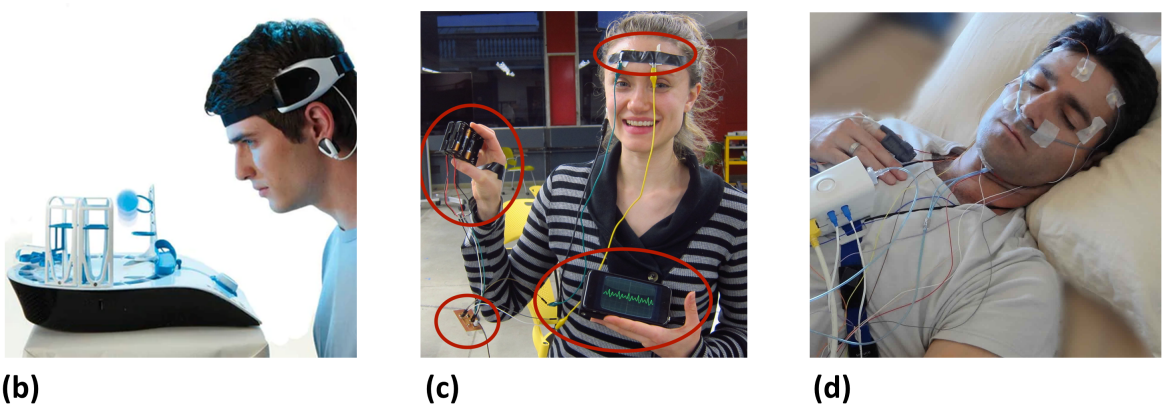


Figure 3.1: (a) Left: Professional EEG cap with many electrodes. Center: Raw EEG signals derived from a selection of electrodes. Right: Exemplars of waveforms with different peak frequencies, along with the cognitive alertness state they are associated with. Figures taken from [10], [11], and [12]. (b) Mindflex, a commercially available game that, too, is controlled by the intensity of mental focus, measured using a mobile EEG headset. Figure taken from [7]. (c) Self-made low-cost portable EEG electrode system. Figure taken from [1]. (d) Depiction of electrode placement in a clinical sleep laboratory, containing far fewer electrodes relative to full-scale EEG caps frequently used in neuroscience laboratories. Figure taken from [9].

First feasibility study and offline analysis pipeline

As we expected the raw EEG data to come with high variability and noise, instead of directly attempting to control the speed of our LEGO vehicle on the basis of these raw signals in an 'online' fashion, we first conducted an 'offline' feasibility study. To this end, one carefully placed the EEG electrodes, using the sticky pads included in the WallySci E3K Bio-Sensing Platform package, on the forehead. The participant was instructed to sit comfortably and move as little as possible in order to avoid any potentially confounding movement artefacts. The participant was recorded in six different settings (see Figure 3.2). Each recording lasted for around 10 minutes. The resulting data were stored on a computer and analyzed as follows: For each recording lasting for approximately 10 minutes, we took non-overlapping 200 ms time windows and performed the Fast Fourier Transform (FFT) on each of them (note that the window size of 200 ms was chosen so as to be long enough to provide enough signal strength and to allow the extraction of frequencies as low as 5 Hz but to not be too long and render the later 'online' game experience too slow). We repeated the application of the FFT for each of the non-overlapping 200 ms recordings and, following the removal of trials with poor signal quality (i.e., those whose maximum amplitudes exceeded a certain threshold), we plotted the mean and standard deviation of the frequency response for each of the six recorded conditions (see Figure 3.2).

Figure 3.2: Separately for each of the six recorded conditions: Means and standard deviations of the frequency responses for thousands of non-overlapping 200 ms windows, as extracted from the raw recordings spanning total durations of approximately 10 minutes each.

As is evident in Figure 3.2., there are marked differences across the different experimental conditions, aligned with our previously presented account of cognitive alertness. EEG signals in settings that would be expected to induce more focus, such as proofreading a paper or carrying out math calculations in a flashcard-like manner, were characterized by the stronger presence of very high frequencies (here, in the 25 Hz range) than settings that would be expected to induce sleepiness or relaxation (such as trying not to think, having the eyes closed and relaxing, or having the eyes closed and almost falling asleep).

As our final objective is to differentiate between being very focussed and being very relaxed, we carried out a pilot analysis to determine the statistical strength with which such differentiation can take place when simply considering the differential power that is present in the very high frequencies (here, at 25 Hz). To this end, we extracted the power values present at 25 Hz for each of the 200 ms windows belonging to the recordings from the 'proofreading a paper' and 'eyes closed and almost sleeping' conditions. This analysis revealed strongly statistically significant differences between the means of the two data sets ($p < .0001$ for two-samples t-test) as well as a large effect size (Cohen's $d = 1.4$). Based on this effect size and using the software GPower, we performed a power analysis to provide rough guidance on how many samples of 200 ms windows, on average, would be needed to observe with a certain power (here, a 90 percent probability) a significant difference between the two settings (here, while having a false positive rate of maximally 5 percent). This analysis revealed that only 10 samples for each condition, on average, would be needed to be able to differentiate between the two settings. Note that these results are obtained simply on the basis of the power present at a specific frequency. As will be evident from the sections to follow, these encouraging results can be further improved when extracting more complex features.

Final online analysis pipeline

Given the encouraging observations made as part of our first feasibility study, we adapted our pipeline for an 'online' extraction of frequency profiles as follows: after each 200 ms, an FFT is run and the resulting frequency response is stored. As a single sample of a 200 ms window would carry a too low signal-to-noise ratio, we provide a live plot of the > 2 second moving average of the frequency response (i.e., depicting the mean and standard deviation of the 10 last 200 ms windows' FFT responses), updated every 200 ms (see Figure 3.3. for an example). Having 10 samples as size for this moving average is aligned with the results from our previous power analysis and also ensures a dynamic experience for the participant (this way, switches between recognized alertness states would not result

in excessively-long time lags due to very long moving averages).

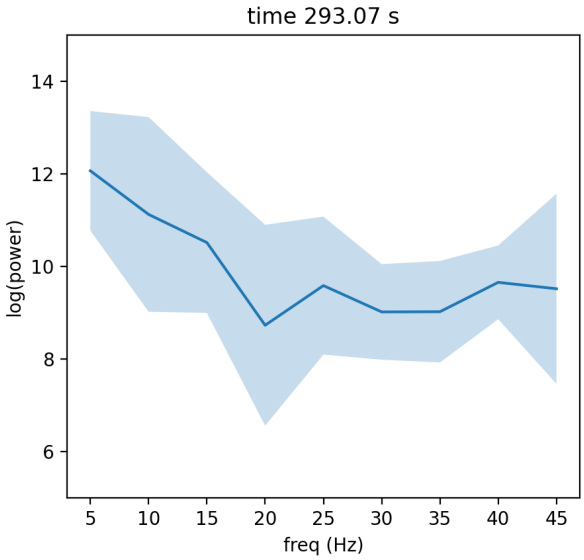


Figure 3.3: Live plot of 2 second moving average of the frequency response: mean and standard deviation of the 10 last 200 ms windows' FFT responses are being depicted.

With this setup, we carried out several online recordings with several different participants. These experiments revealed some inter-individual differences with regard to the specific frequency values that provided the most diagnostic and stable information for differentiating between alertness states. This necessitated a better approach for feature selection than simply comparing powers at a fixed frequency value. After having explored several metrics mapping frequency response to cognitive alertness with limited success, we decided to move towards a more advanced and individualized approach. Specifically, instead of defining a metric ourselves, we chose to have a Support Vector Machine (SVM) 'learn' such metric, for each participant individually, on the basis of frequency responses that are extracted during an approximately 2 minute calibration / training phase.

This training phase takes place as follows: After applying the electrodes to the participant, the participant looks at a screen depicting the word 'focus' for 40 times 200 ms (more than 8 seconds since there is some delay between the measurements), followed by the word 'relax' for 40 times 200 ms, both repeated 3 times. As quickly switching between the two cognitive states proved to be challenging, we decided to only use the latter half (i.e., the last 20 out of 40 samples) for each of the 6 short calibration sections. This provides us with a total of 30 seconds of 'relaxed' EEG signals and 30 seconds of 'focussed' EEG signals. For each condition, the FFT was applied to 60 segments of 200 ms. These outputs of the FFT were subsequently fed into an SVM, with the 150 data points for 'focused' labeled as 0, and the 150 data points for 'relaxed' labeled as 1. The SVM is then trained on learning to differentiate the two.

Following training of the SVM, the main experiment (or 'game') begins: the participant is now can control the speed of the moving car (or, equivalently, the plotted output of the 'focus score' or speed graph) by making oneself in 'focused' state or 'relaxed' state as the participant did in the training phase. This final score is updated every 200 ms, as follows:

1. Every 200 ms, a new FFT is computed.
2. The FFT responses for the last 10 time windows are averaged.
3. This average is fed into the SVM.
4. The SVM provides the predicted label 'focussed' or 'relaxed'
5. With every predicted label 'focused', the speed of the car increases (until reaching a ceiling level of speed); with every predicted label 'relaxed', the speed of the car decreases (until reaching a floor level of speed)

With this setup, we carried out several recordings on several participants and were happy to observe that the pipeline worked satisfactorily. Figure 3.4. depicts exemplar temporal progressions of the focus score recorded as part of the experiment. Listings 3.1. - 3.3. depict the entire pipeline.

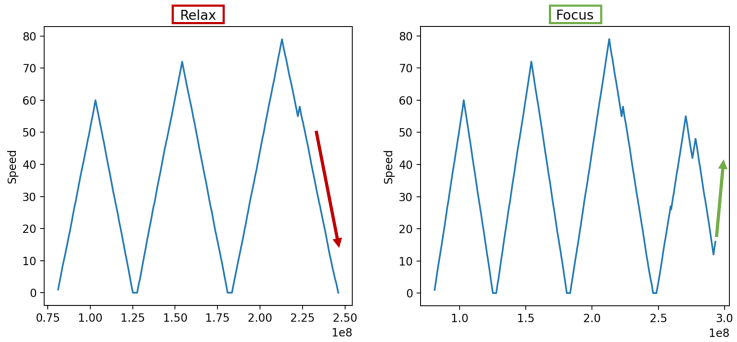


Figure 3.4: Temporal development of the focus score, controlling the speed of the car, when a participant was trying to alternate 'relax' and 'focus' to draw a zigzag pattern. Left: stable and long-lasting reduction in speed when the participant is in 'relaxed' phase. Right: the beginning of an increase in speed, just after the participant switched from 'relax' to 'focus'.

Listing 3.1: [Arduino code for demo]

```
//For recording eeg data
//To be used with record_eeg.py
unsigned long now, prev;

void setup() {
  Serial.begin(115200);
  prev = micros();
}

void loop() {
  // put your main code here, to run repeatedly:
  now = micros();
  if(now - prev >= 2000){
    prev = now;
    Serial.print(now);
    Serial.print(',');
    Serial.print(analogRead(0));
    Serial.print('\n');
  }
}
```

Listing 3.2: [Plotting demo scipt]

```
#importing libraries
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import serial
import sys
import numpy as np
from sklearn import svm

#Base line parameters
num_trials = 3
num_types = 2
list_types = ['Relax', 'Focus']
pred_types = ['Relax', 'Neutral', 'Focus']
num_iter_per_type = 40
baseline_frames = num_trials * num_types * num_iter_per_type

#Speed change
delta_speed = 1

#figure
fig, ax = plt.subplots(2,1, figsize=(5,10))
```

```
ax1 = ax[0]
ax2 = ax[1]
line1, = ax1.plot([],[],'b-')

length = 100
fftfreq = np.fft.fftfreq(length,d=2/1000)[1:10] #data interval is 10ms
data = np.zeros((length,))

#serial port
sensor = serial.Serial('COM3', 115200) #port name and baud
sensor.flushInput()
signal = sensor.readline()

num_samples = 10
fft_samples = np.zeros((len(fftfreq), num_samples))

#svm instance
svm_classifier = svm.SVC()
X = []
y = []

for i in range(num_samples):
    #read in new data
    sensor.flushInput()
    signal = sensor.readline()
    for j in range(length):
        signal = sensor.readline()
        [t, data[j]] = [float(x) for x in signal.decode().split(',')]

    #fft
    fft_samples[:, i] = np.log(np.abs(np.fft.fft(data))**2)[1:10]

mean = np.mean(fft_samples, axis=1)
std = np.std(fft_samples, axis=1)

ax1.clear()
ax1.plot(fftfreq, mean)
ax1.fill_between(fftfreq, mean+std, mean-std, alpha=0.25)
ax1.set_xlabel('freq (Hz)')
ax1.set_ylabel('log(power)')
ax1.set_title("time %.2f s"%(t/1000000))

time_array = []
ind_array = []

def animate_baseline_aquire(i):
    mode = (i//num_iter_per_type)%num_types
    ax2.set_title(list_types[mode])
    ax2.set_xlabel(i)

    #read in new data
    sensor.flushInput()
    signal = sensor.readline()
    for j in range(length):
        signal = sensor.readline()
        [t, data[j]] = [float(x) for x in signal.decode().split(',')]

    #fft
    fft_samples[:, i%num_samples] = np.log(np.abs(np.fft.fft(data))**2)[1:10]

    mean = np.mean(fft_samples, axis=1)
    std = np.std(fft_samples, axis=1)

    if (i%num_iter_per_type)>=20:
        #X.append(fft_samples[:, i%num_samples])
        X.append(mean)
        if mode == 0:
            y.append(-1)
        else:
            y.append(1)
```

```
    #replotting
    ax1.clear()
    ax1.plot(fftfreq, mean)
    ax1.fill_between(fftfreq, mean+std, mean-std, alpha=0.25)
    ax1.set_ylim([5, 15])
    ax1.set_xlabel('freq (Hz)')
    ax1.set_ylabel('log(power)')
    ax1.set_title("time %.2f s"%(t/1000000))


def animate(i):
    if i < baseline_frames:
        animate_baseline_aquire(i)
    elif i == baseline_frames:
        svm_classifier.fit(X, y)
    else:
        #read in new data
        sensor.flushInput()
        signal = sensor.readline()
        for j in range(length):
            signal = sensor.readline()
            [t, data[j]] = [float(x) for x in signal.decode().split(',')]

        #fft
        fft_samples[:, i%num_samples] = np.log(np.abs(np.fft.fft(data))**2)[1:10]

        mean = np.mean(fft_samples, axis=1)
        std = np.std(fft_samples, axis=1)

        pred = svm_classifier.predict([mean])[0]

        if len(ind_array)==0:
            speed = pred * delta_speed
        else:
            speed = ind_array[-1] + pred * delta_speed

        if speed > 255:
            speed = 255
        elif speed < 0:
            speed = 0

        time_array.append(t/1000000)
        ind_array.append(speed)

        #replotting
        ax1.clear()
        ax1.plot(fftfreq, mean)
        ax1.fill_between(fftfreq, mean+std, mean-std, alpha=0.25)
        ax1.set_ylim([5, 15])
        ax1.set_xlabel('freq (Hz)')
        ax1.set_ylabel('log(power)')
        ax1.set_title("time %.2f s"%(t/1000000))

        ax2.clear()
        ax2.plot(time_array, ind_array)
        ax2.set_title(pred_types[pred+1])
        ax2.set_ylabel('Speed')

try:
    ani = animation.FuncAnimation(fig, animate, interval=0)
    plt.show()
except KeyboardInterrupt:
    plt.close()
    sensor.close()
    sys.exit()
```

Listing 3.3: [Function for car system (no plotting)]

```
#importing libraries
import numpy as np
from sklearn import svm
```

```
#Global Parameters and Variables
#Base line parameters
list_types = ['Relax', 'Focus']
num_trials = 3
num_iter_per_type = 40
baseline_frames = num_trials * len(list_types) * num_iter_per_type

#Speed change delta
delta_speed = 1

#fft related
length = 100
fftfreq = np.fft.fftfreq(length,d=2/1000)[1:10] #data interval is 2ms

num_samples = 10 #number of frames for mean
fft_samples = np.zeros((len(fftfreq), num_samples))

#svm instance
svm_classifier = svm.SVC()
X = []
y = []

#Speed array
speed_array = []

def baseline_acquisition(i, data):
    mode = (i//num_iter_per_type)%len(list_types)
    print("Base line acquisition: " + list_types[mode], end="\r")

    #fft calculation
    fft_samples[:, i%num_samples] = np.log(np.abs(np.fft.fft(data))**2)[1:10]

    #Only use latter half of the data
    if (i%num_iter_per_type)>=num_iter_per_type//2:
        mean = np.mean(fft_samples, axis=1)
        X.append(mean)
        y.append(mode)

    speed_array.append(0)


def get_speed(data):
    assert data.shape==(length,), "data size does not match"

    i = len(speed_array)

    #When the base line data acquisition is done
    #Train svm
    if i == baseline_frames:
        svm_classifier.fit(X, y)
        print("Base line done!", end="\r")

    #At first ground truth data collection
    if i < baseline_frames:
        baseline_acquisition(i, data)
    else:
        #fft calculation
        fft_samples[:, i%num_samples] = np.log(np.abs(np.fft.fft(data))**2)[1:10]
        mean = np.mean(fft_samples, axis=1)

        #Obtain the prediction and update speed
        pred = svm_classifier.predict([mean])[0]
        if pred == 0:
            speed = speed_array[-1] - delta_speed
        else:
            speed = speed_array[-1] + delta_speed

        #Limit the range of the speed (0 to 255 inclusive)
        if speed > 255:
            speed = 255
```

```
        elif speed < 0:
            speed = 0

        speed_array.append(speed)
        print("Current state: " + list_types[pred] + ", Speed: " + str(speed), end= "\r")

    return speed_array[-1]
```

Replicating online analysis pipeline with self-made pad replacements

Given the limited number of electrodes with sticky pads that were delivered as part of the WallySci E3K Bio-Sensing Platform, we constructed our own low-cost electrodes (see Figure 3.5.(a) and 3.5.(b)), connected them to the Bio-Sensing Platform, and repeated the final online analysis described in the previous section. While we found that the overall power display differed marginally from the plot obtained with the previously-used electrodes, the SVM-based speed score analysis worked just as well as with the previous setup (see Figure 3.5.(c)).

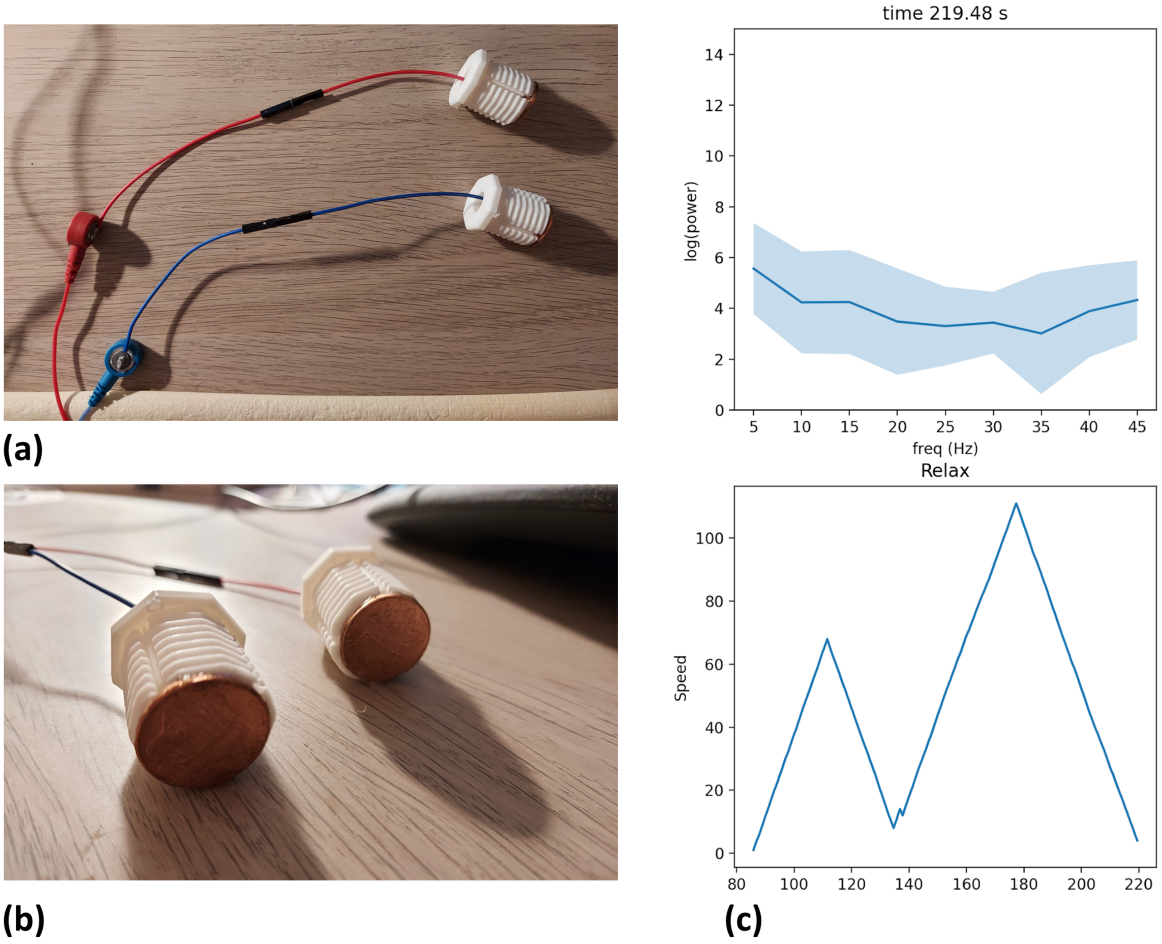


Figure 3.5: (a) and (b) Self-made electrodes made of dimes. (c) Live demo with self-made electrodes. Top: 2 second moving average of the frequency response (mean and standard deviation of the 10 last 200 ms windows' FFT responses) when relaxed. Bottom: Temporal evolution of the focus score when the participant was trying to 'relax' and 'focus' alternately (here, at the end of the 'relax' state).

Electromyography

Electromyography (EMG) allows for the recording of electrical signals originating from muscles. Here, we use non-invasive surface EMG electrodes (delivered as part of the WallySci E3K Bio-Sensing Platform) and apply them to the arm of our participant to detect voluntary muscle contractions. The integration of EMG signals into our design is intended to allow the participant to switch the direction of the

LEGO vehicle from moving forward to moving backward, and vice versa.

As opposed to EEG signals, which measure very subtle differences of brain signatures, EMG signals come with a high signal-to-noise ratio during muscle contractions and are relatively straightforward to analyze. As is evident in Figure 3.6., when a strong movement is elicited, the recorded amplitude exceeds the baseline amplitude by far. Thus, we can simply apply a threshold operation (here, at an amplitude values of 400) and have crossing such threshold translate to switching the direction of the car. Note that after switching the direction of the car, we also implemented a 2-second blockage as the muscles take some time to relax, so as to avoid repeated back-and-forth direction flips. The essential code is provided in Listing 3.4.

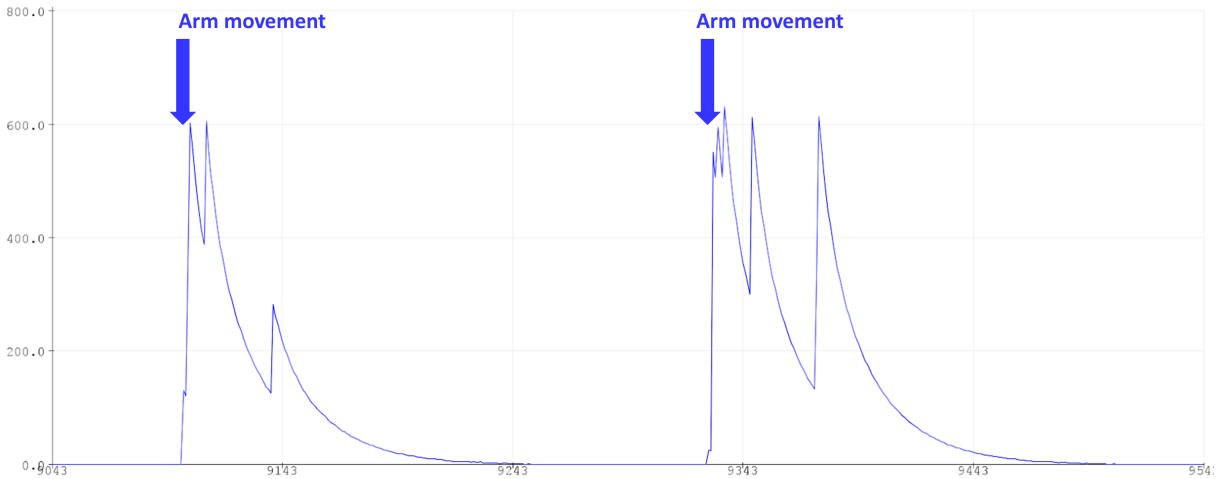


Figure 3.6: Exemplar EMG signal when the arm is relaxed vs. when two arm movements are elicited.

Listing 3.4: [Arduino code for EMG sensor testing].

```
//Unit testing EMG
//Lights up an LED when you make a movement
//Tested on Arduino Uno
int v;
int threshold = 400;
int on;

unsigned long prev;

int pinNumber = 0; //A0 PIN

int inactivePeriod = 2000;

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

void loop() {
  // put your main code here, to run repeatedly:
  v = analogRead(pinNumber);
  Serial.println(v);
  if(v > threshold && on == 0){
    on = 1;
    prev = millis();
  }else if(on == 1 && millis()-prev>=inactivePeriod){
    on = 0;
  }
  if(on){
    digitalWrite(LED_BUILTIN, HIGH);
  }else{
    digitalWrite(LED_BUILTIN, LOW);
```

```
  }
  delay(10);

}
```

Inertial Measurement Unit

The Inertial Measurement Unit (IMU), also provided as part of the WallySci E3K Bio-Sensing Platform, and attached to our 3d-printed EEG headset, allows measuring movements of the head. In our final setup, this allows the participant to control the direction of the wheels of the car. Similar to the extraction of the EMG signals, the analysis pipeline is very straightforward: if the participant tilts the head far to the left (as indicated by the relevant sensor reaching a negative value below a threshold; here, minus 25), the car will move further to the left; if the participant tilts the head far to the right (as indicated by the relevant sensor reaching a positive value exceeding the threshold; here, 25), the car will move further to the right. An exemplar recording is provided in Figure 3.7, and the essential code is provided in listing 3.5.

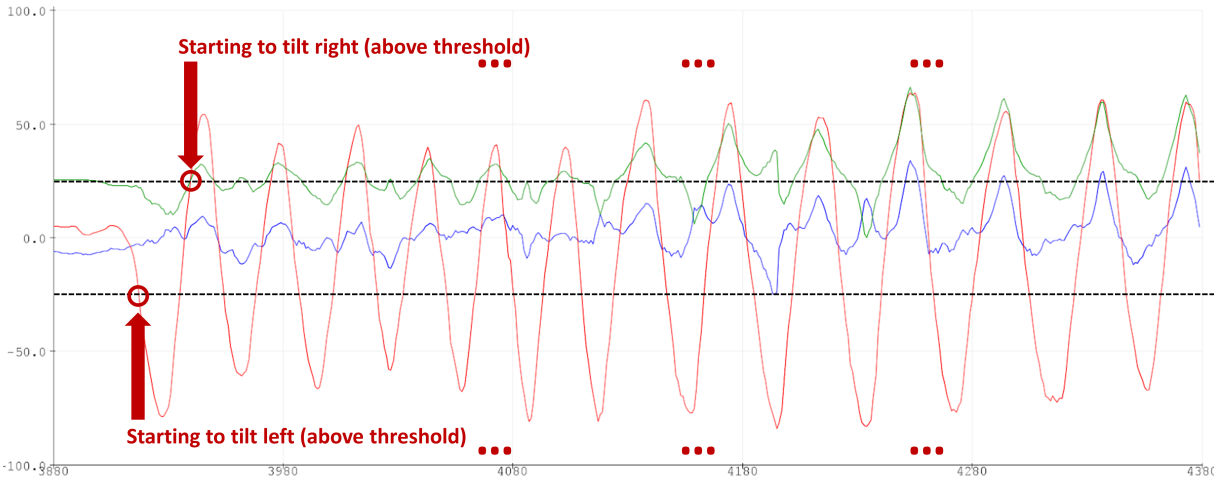


Figure 3.7: Exemplar IMU signal when alternately tilting left and right.

Listing 3.5: [Arduino code for IMU sensor testing].

```
//IMU unit testing
//Tested with E3K
//Lights up different LEDs depending on the tilt.

///////////-----IMU
#include <Wire.h>
#include "SparkFun_BNO080_Arduino_Library.h"
BNO080 myIMU;
boolean isIMU = false;

///////////-----Button and LEDs
int led1 = 4, led2 = 15;
int but1 = 12, but2 = 13;

int threshold = 25;

void setup() {
  Serial.begin(115200);
  Serial.println("The device started");

  ///////////-----Button and LEDs
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
  pinMode(but1, INPUT);
  pinMode(but2, INPUT);
```

```
  ////////////-----IMU

  Wire.begin();
  isIMU = myIMU.begin();
  if(isIMU == false){
    Serial.println("BNO080 not detected at default I2C address. IMU not connected or check
        your jumpers and the hookup guide.");
    Serial.println("Continuing....");
  }else{
    Wire.setClock(400000); //Increase I2C data rate to 400kHz
    myIMU.enableRotationVector(50); //Send data update every 50ms
    Serial.println(F("IMU enabled"));
    Serial.println(F("Rotation vector enabled"));
    Serial.println(F("Output in form i, j, k, real, accuracy"));
  }
}

void loop() {

  ////////////-----IMU
  if(isIMU == true){
    //Look for reports from the IMU
    if(myIMU.dataAvailable()){
        float roll = (myIMU.getRoll()) * 180.0 / PI; // Convert roll to degrees
        float pitch = (myIMU.getPitch()) * 180.0 / PI; // Convert pitch to degrees
        float yaw = (myIMU.getYaw()) * 180.0 / PI; // Convert yaw / heading to degrees

        Serial.print(roll, 1);
        Serial.print(F(","));
        Serial.print(pitch, 1);
        Serial.print(F(","));
        Serial.print(yaw, 1);

        Serial.println();

        digitalWrite(led1, pitch > threshold);
        digitalWrite(led2, pitch < -threshold);
    }
  }

  delay(10);
}
```

The code can be found in the project [GitHub repository].

4

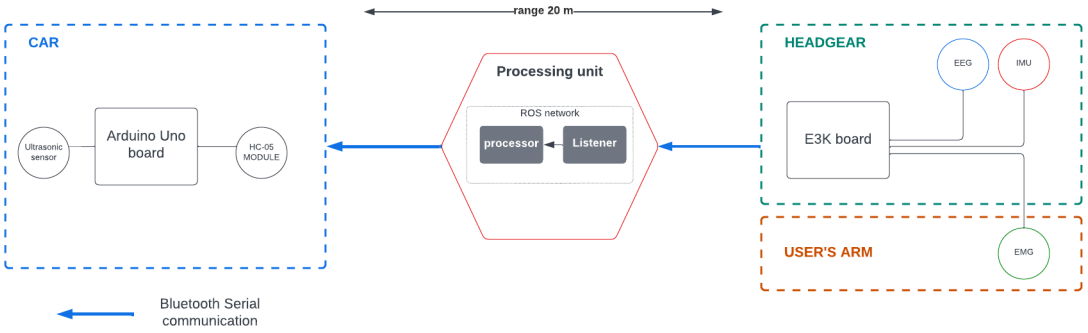
Car Control and Communication

Synopsis

github repository: $https://github.com/EPFL-EEG-Team/robot_mind_control$
The system design is based on the Robot Operating System (ROS) and bluetooth serial communication between all 3 devices: the ESP32 board on the headgear, the Arduino UNO on the car and the central processing unit which can be any computer.
The signals from the electrodes and sensors will be sent to the processing unit in a one-way channel. The processing unit is composed of two modules: the listener which will retrive data and multiplex it and the processor which will process all of the three EEG, EMG and IMU signals in order to encode them into controls for the car. As stated in the introduction, the EEG signal will be used to control the speed of the car. Hence the goal is to classify it with respect to two labels: Focused, Relaxed. The car will increase its speed as the user is focused and decrease it when relaxed. This is accomplished through the use of a Support Vector Machine, as detailed in the previous section. The processing of both EMG and IMU signals is more straightforward (see previous section for details).

Software Architecture

This is an illustration of the architecture discussed above. Let's now dive into the implementation.



Implementation

Requirements

1. Programming Languages:
   (a) python
   (b) C / C++ (for embedded systems sketches)
2. Libraries:
   (a) python libraries:
      i. rospy (included with the installation of any ROS distribution*)
      ii. pyserial (for bluetooth serial communication)
      iii. time
      iv. numpy
      v. struct (to convert data to bytes)
      vi. sklearn (for SVM)
   (b) C / C++ libraries:
      i. <BluetoothSerial.h> (for ESP32 bluetooth serial communication)
      ii. <SoftwareSerial.h> (for bluetooth communication through HC-05 module on Arduino)

Embedded Systems

1. $main - car.ino$

This sketch will be responsible for the control of the car. It will retrieve encoded data from the processor via bluetooth serial and provides inputs to the motor controller and the servo motor accordingly.

```
/**
 * @file main.ino
 *
 * @author Emile Janho Dit Hreich (emile.janhodithreich@epfl.ch)
 *         Michael Freeman (michael.freeman@epfl.ch)
 *
 * @brief The main arduino script to control the car through eeg, emg an
 *
 * @version 0.1
 * @date 2022-04-22
 *
 * @copyright Copyright (c) 2022
 *
 */

// ================================================================================
// LIBRARIES
#include <PWMServo.h>
//#include <Servo_Hardware_PWM.h>
#include <SoftwareSerial.h>
#include <stdio.h>

// ================================================================================
// MACROS AND SETUP

// ---------------------------------------
// motor

// enable pin of motor controller must have pin with pulse modulation
#define MOTOR_EN_PIN  11
#define MOTOR_IN1  13
#define MOTOR_IN2  12
int MOTOR_SPEED = 0;

// ---------------------------------------
// EMG
enum EMG_State {
```

```
  STOP, FORWARD, BACKWARD
};

EMG_State state = FORWARD;

// ---------------------------------------
// Bluetooth

#define TX 3
#define RX 2
#define SERIAL_BAUDRATE 2400
#define BLUETOOTH_BAUDRATE 2400

SoftwareSerial Bluetooth(TX, RX);

// ---------------------------------------
// Serial data

String rawData = "";
int data;
String mode;
String prevMode;
char read                    = 0;
int r = 0;
int valid_data = 0;
// format: EEG_XXXXXXXX
char str_data_retrieved[13] = {0};

// ---------------------------------------
// servo

int servo_pin = 9;
int servo_straight = 90;
int servo_right    = 0;
int servo_left     = 180;

PWMServo servo;

// ================================================================================



void setup() {

    Serial.begin(115200);
    Serial.println("salut monde");
    Bluetooth.begin(9600);

    // motor
    pinMode(MOTOR_EN_PIN, OUTPUT);
    pinMode(MOTOR_IN1, OUTPUT);
    pinMode(MOTOR_IN2, OUTPUT);

    // servo
    servo.attach(servo_pin);


}

// turns the wheels full right
void turnRight() {
  servo.write(servo_right);
  delay(15);
}

// turns the wheels full left
void turnLeft() {
  servo.write(servo_left);
  delay(15);
}
```

```
void goStraight() {
  servo.write(servo_straight);
  delay(15);
}

void changeSpeed(int speed) {
  if (speed <= 0) {
    digitalWrite(MOTOR_IN1, LOW);
    digitalWrite(MOTOR_IN2, HIGH);
    if (speed == 0) {
      MOTOR_SPEED = 0;
    }else if (speed <= -200) {
      MOTOR_SPEED = 200;
    } else {
      MOTOR_SPEED = -speed;
    }
  } else if (speed >= 200) {
    digitalWrite(MOTOR_IN1, HIGH);
    digitalWrite(MOTOR_IN2, LOW);
    MOTOR_SPEED = 200;
  } else {
    digitalWrite(MOTOR_IN1, HIGH);
    digitalWrite(MOTOR_IN2, LOW);
    MOTOR_SPEED = speed;
  }
  analogWrite(MOTOR_EN_PIN, MOTOR_SPEED);
}

void loop() {

  // check if data is available

  if (Bluetooth.available()){

    char a = Bluetooth.read();
    rawData.concat(a);
    r += 1;
    prevMode = String(mode);
    if (r == 13){
      r = 0;
      mode = rawData.substring(0, 3);
      data = rawData.substring(4, 13).toInt();
      rawData = String();

    }

    Serial.println(mode);
    //Serial.println(data);

    if (mode == "EEG") {
//      Serial.println("debug");
      if (state == FORWARD) {
        changeSpeed(data);
      } else if (state == BACKWARD) {
        changeSpeed(-data);
      }
    } else if (mode.equals("EMG")) {
      if (!prevMode.equals("EMG")) {
        if (state == STOP) {
          state = FORWARD;
          digitalWrite(MOTOR_IN1, 1);
          digitalWrite(MOTOR_IN2, 0);
        } else if (state == FORWARD) {
          state = BACKWARD;
          digitalWrite(MOTOR_IN1, 0);
          digitalWrite(MOTOR_IN2, 1);
        } else if (state == BACKWARD) {
          state = STOP;
          changeSpeed(0);
        }
        Serial.println(state);
```

```
      }


    } else if (mode.equals("IMU")) {
      Serial.println("IMU");

      if (data < -20) {
//        Serial.println("turning left");
        turnLeft();
      } else if (data > 20) {
        turnRight();
      } else {
        goStraight();
      }
    }else if (r == 13){

      while(rawData != "EEG" && rawData != "EMG" && rawData != "IMU"){
        rawData = String();
        if (Bluetooth.available()>3){
           rawData.concat(Bluetooth.read());
           rawData.concat(Bluetooth.read());
           rawData.concat(Bluetooth.read());
        }

      }
      r = 3;
    }

  }
}
```

The only libraries to include are $< SoftwareSerial.h >$, $< stdio.h >$ and $< PWMServo.h >$. The SoftwareSerial library will be used to communicate with the rest of the system through the HC-05 Bluetooth module. We use PWMServo library instead of the classical Servo library because the standard library was conflicting with the Bluetooth module, causing the servo to behave unexpectedly.
This sketch contains all the main functions to move the car, such as change the speed of the car and turn the gear to go right, left and straight. In the main loop algorithm, it will receive data transmitted by the head gear and will process it. The message will contain the appropriate data followed by an identifier for which signal is being sent (EEG, IMU, EMG). We have decided to implement 3 states of the car : going forward, backward and stopped. those states are controlled by the EMG signal. Once an EMG signal is received, the state changes. States will loop indefinitely in the following pattern :

$$STOP- > FORWARD- > BACKWARD- > STOP- > ...$$

To avoid changing states multiple times when receiving several consecutive EMG signal when contracting a muscle, we keep track of what was the previous mode received. If it was an EMG signal we simple ignore it, causing the script to consider only the first EMG signal to change the state only once.
The EEG data will be used to control the speed of the car. The data received will be integers that will directly be passed to the changeSpeed() function, accordingly to which state we are in. Going forward will pass positive values, backward will pass negative values and stopped will pass 0.
IMU will serve to move the car right or left. if the data received is beyond a determined threshold the servo will rotate right or left, depending on weather the value is positive or negative. if the value doesn't exceed the threshold, the car will go straight.
If either of those signals were received, we interpret it as a corrupted signal and we reconstruct the signal.

2. $main - headgear.ino$
   This skecth will contain all necessary code to retrieve data from the electrodes and send it to the processing unit.

```
/**
 * @file main.ino
 * @author Emile Janho Dit Hreich (emile.janhodithreich@epfl.ch)
 * @brief The main arduino script to control the car through eeg, emg an
 * @version 0.1
 * @date 2022-04-22
 *
 * @copyright Copyright (c) 2022
 *
 */

// ============================================================================
// LIBRARIES

#include <BluetoothSerial.h>
#include <stdio.h>
#include <Wire.h>
#include "SparkFun_BNO080_Arduino_Library.h"
// ============================================================================
```

$< BluetoothSerial.h >$ differs from the other sketch because here we are programming an ESP32 board with an integrated bluetooth module. There is no need for an extra HC-05 module.
$< stdio.h >$ will be used to convert input to bytes before sending it through the bluetooth communication channel as required.
Finally, $< Wire.h >$ and SparkFun are the libraries that allow you to communicate I2C/TWI devices, in this case, the IMU module on the headset.

Next we define all the macros and data structures that we will need to retrieve data:

```
// ============================================================================
// MACROS and setup

#define BLUETOOTH_BAUDRATE 9600

#define EEG_PIN 33
#define EMG_PIN 39

// ------------------------------------------------
// Bluetooth
BluetoothSerial SerialBT;

// ------------------------------------------------
// EEG

unsigned long now, prev;

float f = 0.0;
char c[50]  = {0}; //size of the number
char type[3]= {0};

int data_points = 0;

// ------------------------------------------------
// EMG

unsigned long now_EMG, prev_EMG;

float value = 0;
char emg[50]  = {0}; //size of the number


// ------------------------------------------------
// IMU
BNO080 IMU;
boolean isIMU = false;

unsigned long now_IMU, prev_IMU;
```

```
float roll  = 0;
float pitch = 0;
float yaw   = 0;

char imu[50]  = {0}; //size of the number
// ================================================================================
```

We first define two macros for serial communication, one defines the baudrate for Bluetooth communication and the second is for debugging using the serial monitor (i.e. data is sent to a local serial port on the computer in which the ESP32 board is plugged in order to visualize it). If you're not familiar with the term "baudrate", it defines the number of signal units per second in data transmission.
The rest of the data structures are used to actually control and prepare data transmission. The EEG signal will be recorded and sent every 2 ms. now and $prev$ variables of type unsigned long will be used to measure this 2 ms interval.
The variable f of type float will hold the recorded value of the EEG signal which will be concatenated to a string held in the array c defined underneath it and then converted to bytes for transmission. The purpose of the $type$ array is to specify to which signal corresponds the data being sent.
The delivery format is the following:

$$< signaltype > - < value >$$

One example would be: "EEG-23.7".

The same principle and format are applied to EMG signal recording.

In addition, for the IMU, we'll have to initialize a BN080 component to communicate with the IMU module as well as three variables that will store values for the pitch, the yaw and the roll from the IMU.

Finally we implement the two traditional functions for ESP32 and Arduino Sketches which are $setup()$ and $loop()$ functions:

```
void setup()
{

    // Bluetooth
    SerialBT.begin("E3K");
    Serial.begin(BLUETOOTH_BAUDRATE);

    // time
    prev = micros();
    prev_EMG = micros();
    prev_IMU = micros();

    // IMU
    Wire.begin();
    isIMU = IMU.begin();
    if(isIMU == false){
      Serial.println("BNO080 not detected at default I2C address. IMU not connected or
          check your jumpers and the hookup guide.");
      Serial.println("Continuing....");
    }else{
      Wire.setClock(400000); //Increase I2C data rate to 400kHz
      IMU.enableRotationVector(50); //Send data update every 50ms
      Serial.println(F("IMU enabled"));
      Serial.println(F("Rotation vector enabled"));
      Serial.println(F("Output in form i, j, k, real, accuracy"));
  }

}
```

```
void loop()
{
  // ----------------------------------------
  // EEG
  now = micros();

  if(now - prev >= 2000){
    Serial.println("EEG");

    f = analogRead(EEG_PIN);
    sprintf(c, "%g", f);
    strcat(c, "_EEG\n");
    SerialBT.print(c);

    prev = now;

  }

  // ----------------------------------------
  // IMU
  now_IMU = micros();

  if(isIMU == true && now_IMU - prev_IMU >= 1000000){
    //Look for reports from the IMU
    if(IMU.dataAvailable()){
       roll  = (IMU.getRoll()) * 180.0 / PI;   // Convert roll to degrees
       pitch = (IMU.getPitch()) * 180.0 / PI;  // Convert pitch to degrees
       yaw   = (IMU.getYaw()) * 180.0 / PI;    // Convert yaw / heading to degrees

       // DEBUG
       Serial.print(roll, 1);
       Serial.print(F(","));
       Serial.print(pitch, 1);
       Serial.print(F(","));
       Serial.print(yaw, 1);

       Serial.println();
       // END DEBUG

       // !! Only considers the pitch but can be adapted to take everything into account
       // it also depends on the placement of the board
       sprintf(imu, "%g", pitch);
       strcat(imu, "_IMU\n");
       SerialBT.print(imu);


    }
    prev_IMU = now_IMU;
  }

  // ----------------------------------------
  // EMG
  now_EMG = micros();
  if (now_EMG - prev_EMG >= 100000){
      Serial.println("EMG");
      value = analogRead(EMG_PIN);
      sprintf(emg, "%g", value);
      strcat(emg, "_EMG\n");
      SerialBT.print(emg);

      prev_EMG = now_EMG;

  }


}
```

There is a specific rate at which each type of signal is sent and it is set using the micros() function to measure time. The prev<Type> is a point of reference in time and now<Type> is the real time. When the difference between these two variable is greater than some period, data is written to

the buffer and a new point of reference in time is saved.

Processing Unit

ROS setup

ROS is the backbone of the processing unit. Consider a robot as a collection of nodes that produce data. Each node is a process that will publish this data to channels called Topics. Other nodes that wish to access this data will subscribe to its respective channel. This functionality called the Publish/-Subscribe model for communication is implemented as a peer-to-peer network.
In this project we have a 4-nodes robot, but only two are actually ROS nodes: the listener and the processor contained in the processing unit. The reason for this is that ROS is too heavy to run on embedded systems, although there exists some light version distributions especially made for embedded systems. We invite you to check them out in the ROS official website.

Listener

The listener, (listener.py file) is the node responsible for listening to the ESP32 board sending signal data, parse it, and distribute it to the different parts of the processor based on the type of data.
In order to communicate with the board, we use the Pyserial library and bind the bluetooth address of the ESP32 board to a listening port on the computer.
Binding the address to a specific port is done differently in Windows and Linux. In windows, you would have to look for the COM ports in the device manager. Locate the name of your remote device and get the port it is communicating through to put it in the COM PORT variable. In Linux, you would have to manually bind some rfcomm port to the address of your remote device. In this project we use rfcomm0 and rfcomm1. You can find the addresses of your devices in the Bluetooth settings by double clicking on the name of the device. Once you get the address, the command to bind it to a port is the following:

```
sudo rfcomm bind <port number> <device address>
```

Here is the code for this part:

```
'''
    @file: listener.py
    @date: 12/05/2022
    @author: Emile Janho Dit Hreich
             emile.janhodithreich@epfl.ch
    @brief:
'''
# ============================================================
# Libraries

import serial           # Pyserial library
import rospy            # ROS Noetic
import time

from std_msgs.msg import Float32MultiArray, Float32, Int32

# ============================================================
# Some constants

COM_PORT   = "COM9"
BAUDRATE   = 115200
ENCODING   = "ascii"
PARSE_CHAR = "_"
# ============================================================
# Functions

def connect():
    '''
        Establishes connection between device and ESP32 Board
    '''
    while (True):
        # will try to listen on specified port (COM_PORT)
        try:
            listener = serial.Serial(COM_PORT, BAUDRATE, timeout = 1)
            print("Connection established")
```

```
            return listener
        except:
            print("Unable to connect to remote device. Retrying...")
            time.sleep(5)

def process_EEG(array):
    EEG_publisher.publish(Float32MultiArray(data=array))

def process_EMG(data):

    EMG_publisher.publish(data)

def process_IMU(data):
    IMU_publisher.publish(data)



EEG_data = []

def retrieve_data(listener):
    '''
        Listens to the ESP32 board on the headgear and call functions
        to prepare its processing
    '''
    # initializing the array that will contain EEG data
    global EEG_data


    while (True):

        # Parse the string of fromat "<Type of data>_<value>"
        if (listener.in_waiting > 0):

            data = listener.readline().decode(ENCODING)
            # print(data)
            (value, msg_type) = data.split(PARSE_CHAR)



            if (msg_type == "EEG\n"):

                # convert string to float
                value = float(value)
                EEG_data.append(value)
                # print(EEG_data)
                if len(EEG_data) == 99:
                    # print("sent")
                    process_EEG(EEG_data)
                    # time.sleep(0.2)
                    listener.flushInput()
                    EEG_data = []


            elif (msg_type == "EMG\n"):
                # convert string to float
                print(value, msg_type)

                value = float(value)
                process_EMG(value)


            elif (msg_type == "IMU\n"):
                # convert string to float
                print(value, msg_type)

                value = float(value)
                process_IMU(value)
```

```
# ============================================================
# Main

if __name__ == "__main__":

    # ROS node initialzation
    rospy.init_node("Listener", anonymous=False)

    # ROS publishers
    EEG_publisher = rospy.Publisher("EEG", Float32MultiArray, queue_size=1)
    EMG_publisher = rospy.Publisher("EMG", Float32, queue_size=1)
    IMU_publisher = rospy.Publisher("IMU", Float32, queue_size=1)

    # Establish connection
    listener = connect()

    retrieve_data(listener)
```

Caveat! The verification in the connect() function checks if the script is able to listen on the specified port and not whether the Bluetooth device is still connected or not. This is definitely something to enhance for future contributions but for the sake of simplicity we chose to keep it this way to present this project.

Once the connection is established, Data will be parsed and published to the processor via ROS topics. The retrieve data will loop and check of incoming data in the serial buffer. As soon as some data is available, it will read it.

The Main of the listener initializes the ROS node for the listener as well as all the required publishers, one for each type of data. It will then connect to the device and retrieve data as long as it's on.

The retrieval of EEG signal is worth detailing here. It is different from the other two because it requires recording for a certain period of time. That is, the listener will wait until it has retrieved a specific number of values, here a hundred before sending the batch to processing. Also, it will flush the input buffer after this operation, before recording the new batch, because any incoming data that came while processing is considered garbage. The reason for this is that the buffer will fill itself much faster than the speed of the processing. Taking into account the garbage data will make the control of the car less precise because it will be, at some point, completely shifted in time.

Processor

The Processor is the second part of the processing unit. It will subscribe to the listener and process data to encode it and provide instructions to the car. The Bluetooth connection is identical to the listener, but we will use it to transmit data to the car.
Let's have a look at the code:

```
'''
    @file: encoder.py
    @date: 12/05/2022
    @author: Emile Janho Dit Hreich
             emile.janhodithreich@epfl.ch
             Marin Vogelsang
             marin.vogelsang@epfl.ch
    @brief: This file contains all processing functions for
            EEG, EMG and IMU data. It is subscribed to the
            listener (listener.py file). Upon reception of
            data, it will process and encode corresponding
            controls for the car. The commands will be sent
            to the car via Bluetooth serial.
'''
# ============================================================
# Libraries

import serial           # Pyserial library
import rospy            # ROS Noetic
import time
import numpy as np
```

```
import struct

from sklearn import svm
from std_msgs.msg import Float32MultiArray, Float32, Int32

# from listener import EEG_data
# ============================================================
# setup

# ----------------------------------------
# Bluetooth
COM_PORT    = "COM12"
BAUDRATE    = 9600

# ----------------------------------------
# EEG
list_types = ['Relax', 'Focus']
NUM_TRIALS = 3
NUM_ITER_PER_TYPE = 40

baseline_frames = NUM_TRIALS * len(list_types) * NUM_ITER_PER_TYPE

#Speed change delta
delta_speed = 5

# Fast-Fourier Transform
length  = 99
fftfreq = np.fft.fftfreq(length, d = 2 / 1000)[1:10] # data interval is 2ms

num_samples = 10                                     # number of frames for mean
fft_samples = np.zeros((len(fftfreq), num_samples))

#svm instance
svm_classifier = svm.SVC()
X = []
y = []

#Speed array
speed_array = []

# ----------------------------------------
# EMG
EMG_BASELINE_ITERATION = 50
EMG_THRESHOLD          = 20      # Difference between basline and recorded value

EMG_baseline_acquisition_table = []
EMG_baseline = -1

EMG_FLAG = 1

# ----------------------------------------
# IMU
IMU_THRESHOLD_R = 0
IMU_THRESHOLD_L = 0

# ============================================================
# Functions

def connect():
    '''
        Establishes connection between device and ESP32 Board
    '''
    while (True):
        try:
            controller = serial.Serial(COM_PORT, BAUDRATE, timeout = 1)
            print("Connection established")
            return controller
        except:
            print("Unable to connect to remote device. Retrying...")
            time.sleep(5)
```

```
def baseline_acquisition(i, data):
    '''Documentation missing
    '''
    mode = (i // NUM_ITER_PER_TYPE) % len(list_types)
    print("Base line acquisition: " + list_types[mode], end="\r")

    # fft calculation
    fft_samples[:, i%num_samples] = np.log(np.abs(np.fft.fft(data))**2)[1:10]

    # Only use latter half of the data
    if (i % NUM_ITER_PER_TYPE) >= NUM_ITER_PER_TYPE // 2 :
        mean = np.mean(fft_samples, axis=1)
        X.append(mean)
        y.append(mode)

    speed_array.append(0)

def get_speed(data):
    '''
        data: Array[float] contains data aquired over a period of 2 ms.
        EEG processing function.Trains the SVM, performs calibration with respect to a
        baseline then starts computing the speed of the car. The baseline is acquired
        with baseline_acquisition() function.
    '''
    print(data.shape)
    assert data.shape==(length,), "data size does not match"

    i = len(speed_array)

    # When the base line data acquisition is done
    # Train svm
    if i == baseline_frames:
        svm_classifier.fit(X, y)
        print("Base line done!", end="\r")

    # At first ground truth data collection
    if i < baseline_frames:
        baseline_acquisition(i, data)
    else:
        #fft calculation
        fft_samples[:, i % num_samples] = np.log(np.abs(np.fft.fft(data))**2)[1:10]
        mean = np.mean(fft_samples, axis=1)

        #Obtain the prediction and update speed
        pred = svm_classifier.predict([mean])[0]
        if pred == 0:
            speed = speed_array[-1] - delta_speed
        else:
            speed = speed_array[-1] + delta_speed

        #Limit the range of the speed (0 to 255 inclusive)
        if speed > 255:
            speed = 255
        elif speed < 0:
            speed = 40

        speed_array.append(speed)
        print("Current state: " + list_types[pred] + ", Speed: " + str(speed), end= "\r")

    # print(speed_array[-1])
    return speed_array[-1]



def get_orientation(data):
    '''
        EMG processing function
    '''
    global EMG_baseline
    global EMG_FLAG
    # baseline acquisition
```

```
    if (len(EMG_baseline_acquisition_table) < EMG_BASELINE_ITERATION):
        EMG_baseline_acquisition_table.append(data)
    else:

        if EMG_baseline == -1:
            EMG_baseline = np.mean(np.array(EMG_baseline_acquisition_table))

        # encoding
        if (data - EMG_baseline > EMG_THRESHOLD and EMG_FLAG == 1):
            data = "EMG_00000000\n"
            controller.reset_input_buffer()
            controller.write(bytes(data, 'ascii'))
            controller.flush()
            EMG_FLAG = 0
        elif(data == 0):
            EMG_FLAG = 1



def process_EEG(msg):
    '''
        callback function for EEG topic
    '''

    # global EEG_data

    values = np.array(msg.data)

    speed  = get_speed(values)
    # ============================
    # debug
    # print(speed)
    # ============================

    data = "EEG_" + str(int(speed)).zfill(8) + "\n"
    controller.reset_output_buffer()
    controller.write(bytes(data, 'ascii'))
    controller.flush()


def process_EMG(msg):
    '''
        callback function for EMG topic
    '''
    get_orientation(msg.data)

def process_IMU(msg):
    '''
        callback function for IMU topic
    '''
    # directly send to the car. No prior processing is necessary
    # because the conversion to roll-pitch-yaw happens on the previous layer
    # we could do it here as well.

    # print(str(int(msg.data)).zfill(8))
    data = "IMU_" + str(int(msg.data)).zfill(8) + "\n"

    controller.reset_output_buffer()
    controller.write(bytes(data, 'ascii'))
    controller.flush()



# ============================================================
# Main

if __name__ == "__main__":

    # ROS node initialzation
    rospy.init_node("Processor", anonymous=False)
```

```
# ROS Subscribers
rospy.Subscriber("EEG", Float32MultiArray, process_EEG)
rospy.Subscriber("EMG", Float32, process_EMG)
rospy.Subscriber("IMU", Float32, process_IMU)

# connect to remote device
controller = connect()

# ROS loop
rospy.spin()
```

Signal processing has been detailed in the above sections. We will focus here on the communication system. The difference the in the Main is that we have subscriber and not publishers. This directly follows the idea behind this node: it subscribes to the listener, processes data and controls. the car. The call to rospy.spin() will start an infinite loop that will keep listening to incoming ROS messages.
To understand the architecture of this node, you should be familiar with the ROS publish/subscribe model for communication. The important aspect here is the callback functions for the subscribers. Each subscribers is assigned a callback function that will be called as soon as a piece of data is received on the channel corresponding to the topic it is subscribed to. In that case we have three callback functions, one for each type of signal.
As soon as processing is done for each signal and that meaningful instructions have been extracted and encoded, you should encode the instructions in bytes under a specific format that can be decoded by the car. The format that we use is close to what we had at the listener level: we first specify the type of signal the data we are sending corresponds to and we concatenate the value associated with it separated by an underscore. That is, we first construct a string that follows this format. In order to have a uniform size, the value is converted to a string with eight characters by adding the necessary leading zeros at the left. Then, the constructed string is converted to bytes using the bytes type casting in python.
Besides, an important thing to take into account is data corruption. The Bluetooth serial communication is unreliable. Hence, before sending any new piece of data to the buffer, we empty it, then, after writing new data, we wait until the entire data was written before allowing further processing or data transmission. This is done with the two functions reset-output-buffer() and flush(). This is a minimal simple way of ensuring that the car receives clean data. Nonetheless, it remains extensively unreliable, but again, for the purpose of this project we chose to stick to this simple design and leave it open for further contributions to enhance this communication protocol.
Finally, some details about the EMG signal. This type of signal can be considered as a pulse which is why we use it to choose the direction of the car and stop it. However that pulse is high for a period of time that is too high compared to the speed at which data is arriving at the processor. This has the effect that the car will keep switching between STOP, FORWARD, BACKWARD as this pulse is high, whereas the desired functionality is that as soon as the pulse is detected, no matter how long it is, only one switch of this kind occurs. Hence, the flag is set to 0 as soon as the EMG pulse is detected and set back to 1 when the signal goes back down to 0 to allow further detection of EMG.

5

DIY EEG circuit

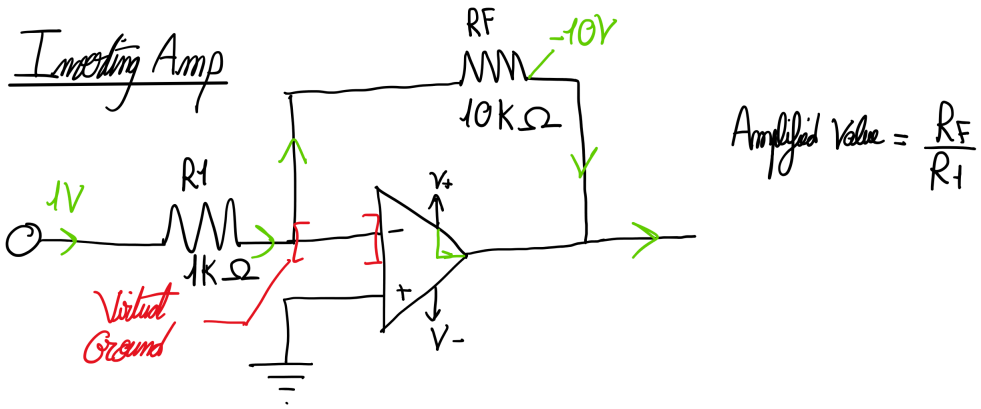
Prelude to the circuit

The DIY circuit consists in a nutshell of a reception, filtering and amplification of the signal emitted by a heart beat, muscle or brain impulses.
These signals have specific frequencies that one can ”capture” and exploit them using Amplifiers, capacitors, resistances and batteries.
This is what we managed to do using these components !
But before getting into the depth of the circuit first, a bit of theory.

Electronics requirements to build an EEG circuit

The power of Amplifiers

Amplifiers are in a nutshell devices that can increase the amplitude of a signal delivering a larger signal that contains all the initial waveform features mainly the frequency.
Amps obey two rules : no current flows in or out of the Amp and The Amp tries to keep the voltage of the inputs the same if a closed loop occurs.
As we can see in the example below:



as a result of this inverting Amp, we obtain a higher voltage at the end.

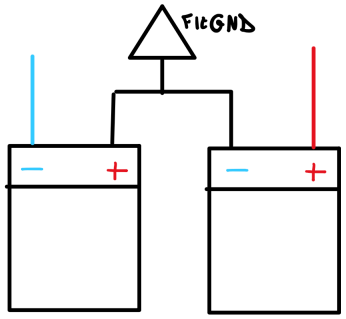
The Creation of Non Polar capacitors

To create a non polar capacitor, one need to connect back to back two polar capacitors in series (The plus outside the minus inside giving + - - +).
They are useful in circuit in which direction is not predefined like in Alternating current circuit.
Watch out for the total capacity of the newly created structure, it is of :

$$C = \frac{1}{\frac{1}{C1} + \frac{1}{C2}} \tag{5.1}$$

The efficacity of a floating ground

In Amps as in connection of two batteries in series the term virtual ground or floating ground (fltGND/ VGND) means that the voltage at the intersection of the two batteries or the intersection of a loop is almost equal to ground voltage (0V).
It is not physically connected to ground.
This helps through a lot of application and helps getting negative volts from a DC battery supply as seen in the schematic below :



The science behind filters

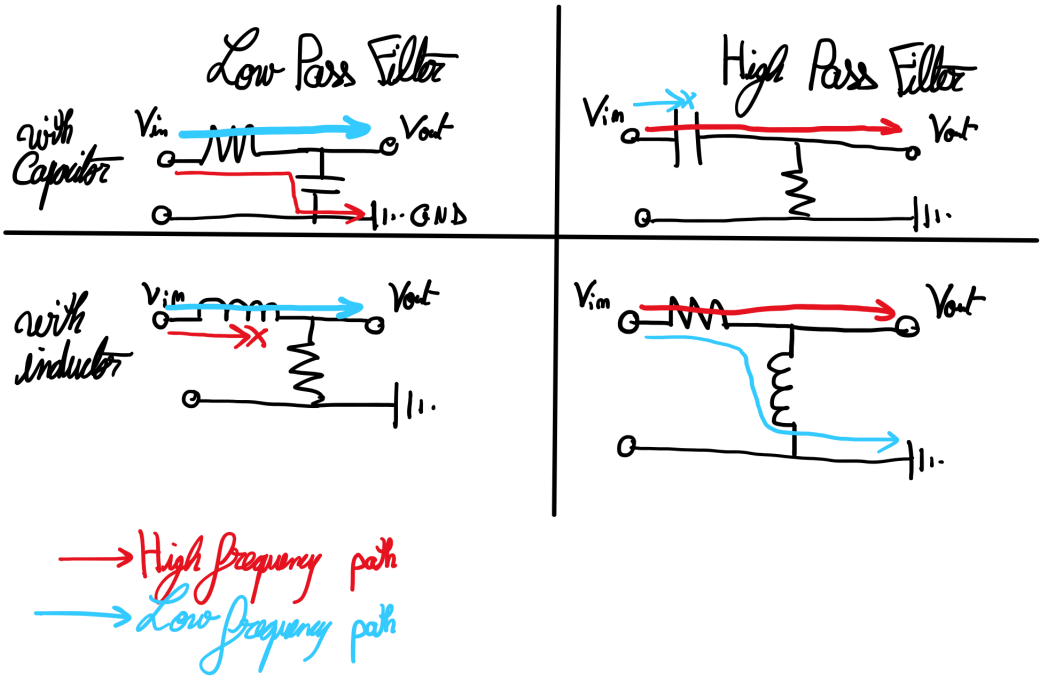
Filters are configurations of capacitors or inductors combined with resistances in order to block some types of frequency. High pass filter let frequencies above the critical frequency pass through it, and blocks frequencies below it or greatly attenuates them. The opposite happens with Low pass filters. Using an inductor, the critical frequency can be calculated using this formula :

$$f = \frac{R}{L * 2 * \pi} \tag{5.2}$$

Whereas using a capacitor, the formula is :

$$f = \frac{1}{R * C * 2 * \pi} \tag{5.3}$$

below you can find a drawing picturing 4 basic filters in RC or RL circuits.



Soldering tips

To solder the different components to the board you will need a soldering iron and some soldering cooper wires at least. some other tools like a cutter or a stand can be practical while soldering.
Before beginning make sure that you all the non-used material are far from the place you are soldering, and don't touch the soldering iron its temperature is around 400 degrees Celsius while you are only at 37 !
After soldering a component don't touch it, it might be very hot. (especially resistances)
To solder components to the board, first insert them in the designated holes and watch out for + and - of the components. Then, turn the board and making sure the tip of the iron is almost perpendicular to the ground, apply the soldering wire into the hole of the component to solder it.
Repeat this process and make sure you don't connect through the conductive Cooper components that should not be connected !

Materials

To build our circuit we used :

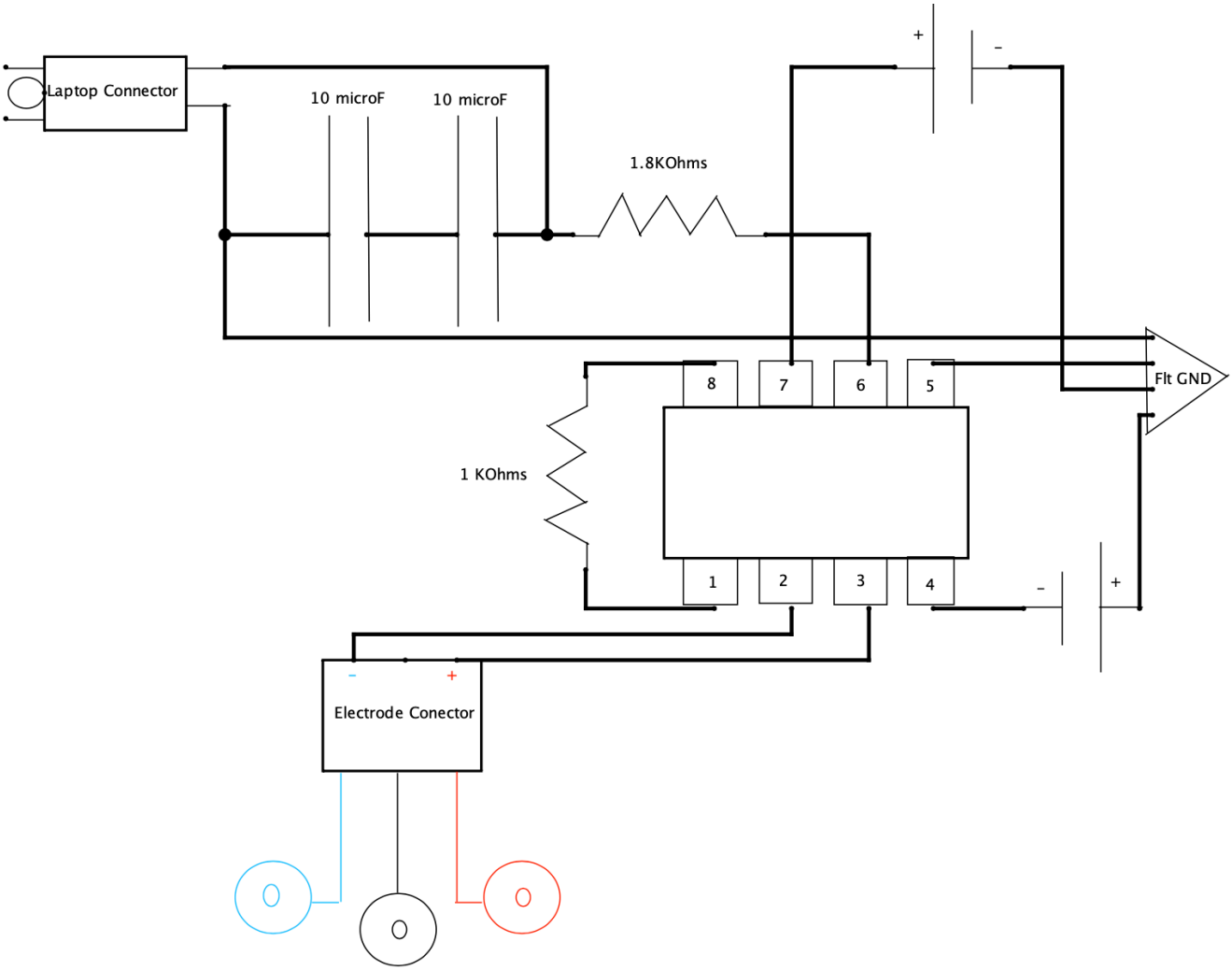
1. A AD620 amplifier
2. A PCB strip board
3. Two Polar capacitors of 10 µF
4. A resistance of 1.8 KΩ
5. A resistance of 1 KΩ
6. Multiple wires
7. Three electrodes (+, - and GND (ground))
8. A laptop connector
9. An Electrode connector
10. Two batteries of 9V

The circuit explained

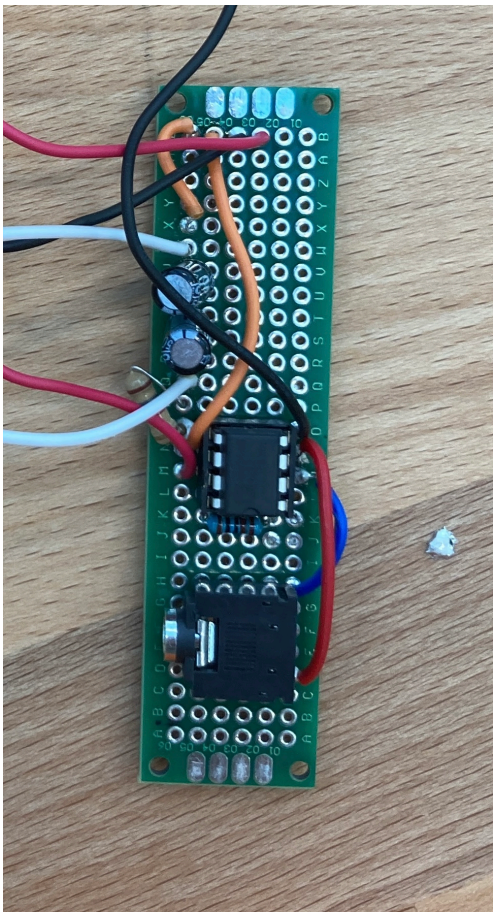
In the circuit below, you can find all the components used to Amplify, filter, and create the current that contains the frequencies we are observing.
This circuit was build and inspired by the Chipstein guide here.
Both batteries are used in order to create the Alternating current the circuit seeks, the 1KΩ resistor and the 1.8KΩ create the amplification using the AD620 amplifier's inputs and outputs.
Both Polar capacitors of 10 µF and the resistance of 1.8KΩ are used to form a low pass filter which removes all frequency above 17.6Hz, 17.6Hz representing the critical frequency of this low pass given by the equation :

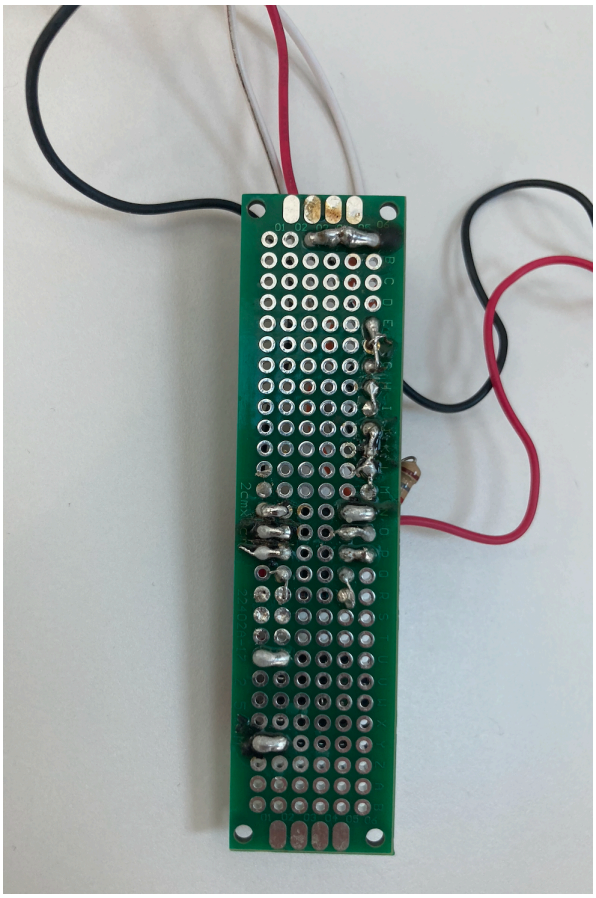
$$f = \frac{1}{R * C * 2 * \pi} \tag{5.4}$$

Concerning the amplifier, the 2 and 3 ports correspond respectively to the negative and positive inputs. The port number 6 represent the output given through the amplification and the port 5 the floating ground of the component.



Here you can see a final version of the circuit :





6

Headset

Designing a 3D headset

The headset was a completely different design compared to the previous sections. It involved less electronics and computer science but more ”do-it-yourself” skills. We decided not to start from scratch but instead looked for a previously tested design on the internet that we were going to adapt.
We decided to take as a basemodel the Ultracortex Mark III Nova Revised Medium Size [13].

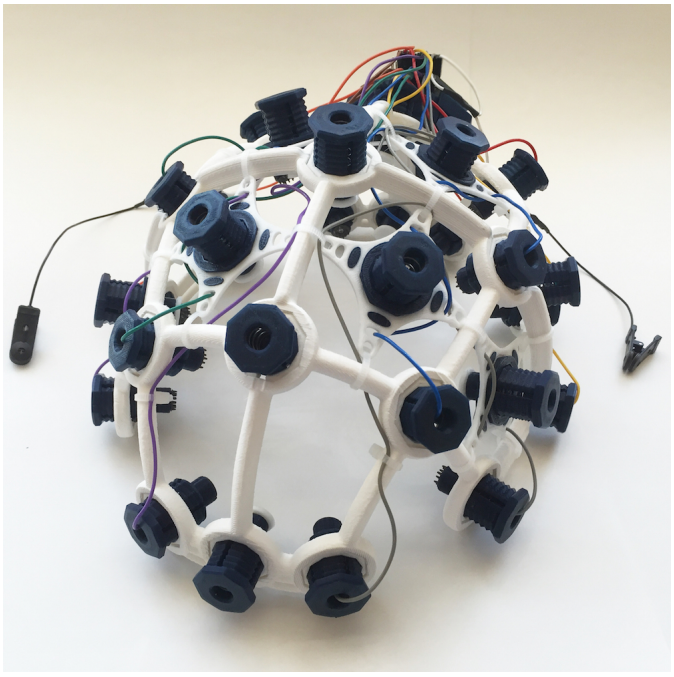


Figure 6.1: Original Headset

As we see in the picture above, EEG nodes are placed inside the screw. We decided not to follow this and to allow the cable to move freely to have more flexibility.
Another adaptation that was required was to redesign a new board holder because we have different needs compared to the original version.

Printing and Assembly of a 3D headset

The first step was to decide and schedule the printing process. We had to plan how and when to print each piece, because we didn’t have unlimited access to the printing room.

First we print the four quarters of the headset. It took around 10h per part. Then we glued them with hot glue.

Figure 6.2: Glueing of Headset

To allow the headset to fit various head shapes, we printed screws and nuts for size adjustment. There were 21 of each and once printed we had to glue the nuts individually to the headset.

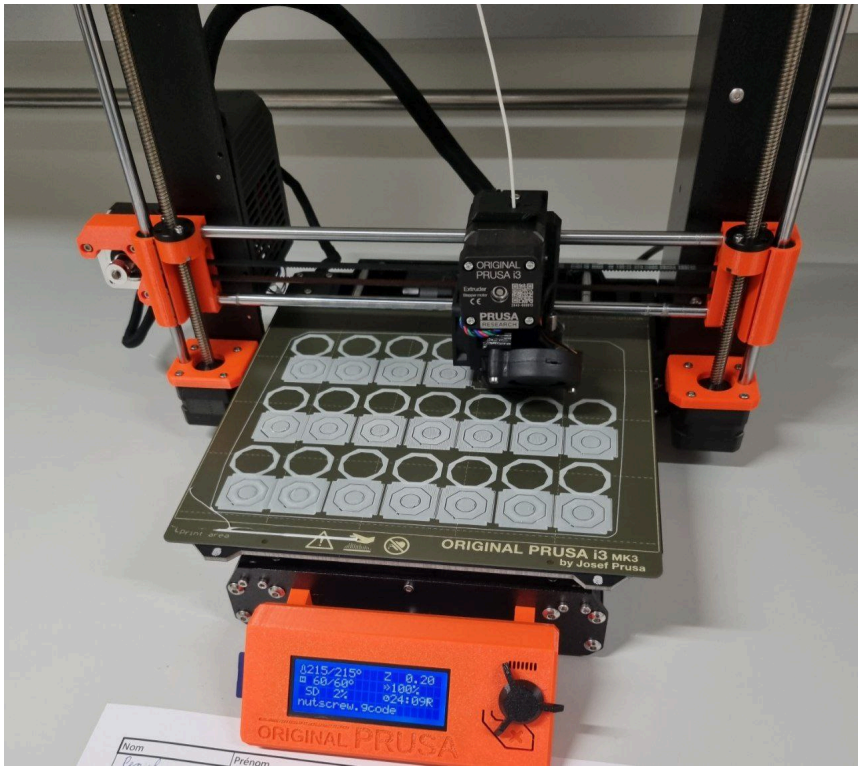


Figure 6.3: Printing process of Screws and Nuts

Board holder design

We then had to design a board holder to hold the electronics above the headset. The main difficulty came from the fact that the electronics didn't have online documentation with the official size of the components. There was also no online 3D model of the E3K board. We decided to draw a stencil of the card, then measured it. This method lacked of precision and therefore we had to print the board holder twice.

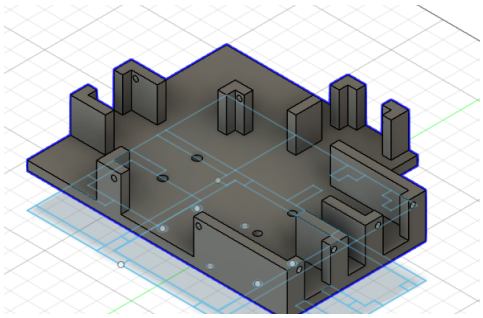


Figure 6.4: Boardholder

Final Assembly

The last part was to attach the boardholder to the headset. Then we placed the board inside it and filed the border in order to be sure that everything fitted in. Then we proceed

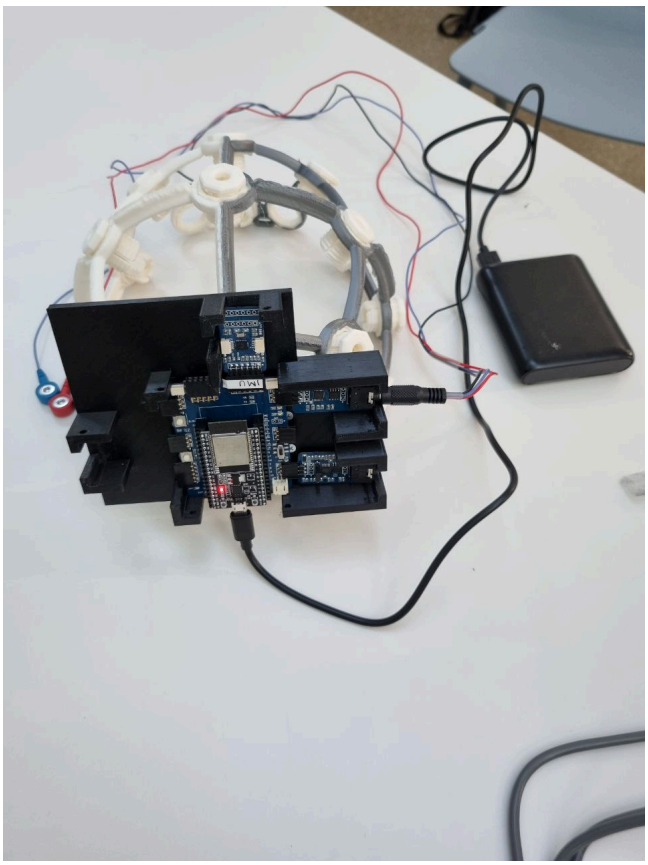


Figure 6.5: Final Headset Design

To find the 3D model described in this chapter, please visit [this github repository]

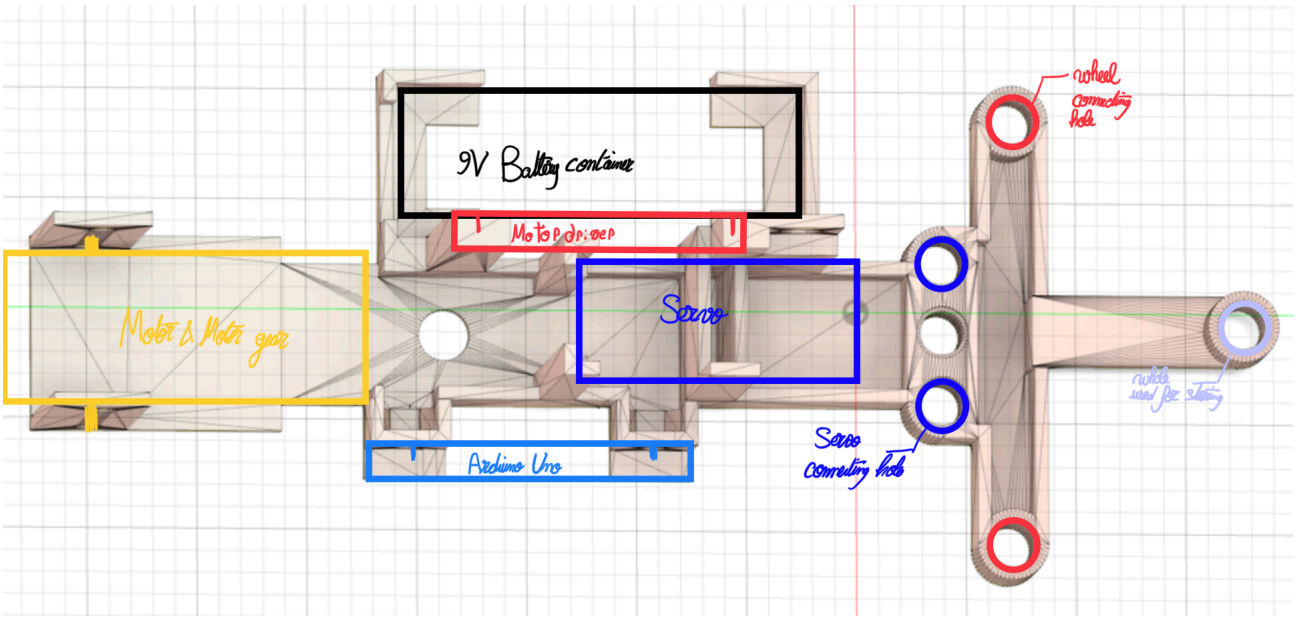
7

Car Design

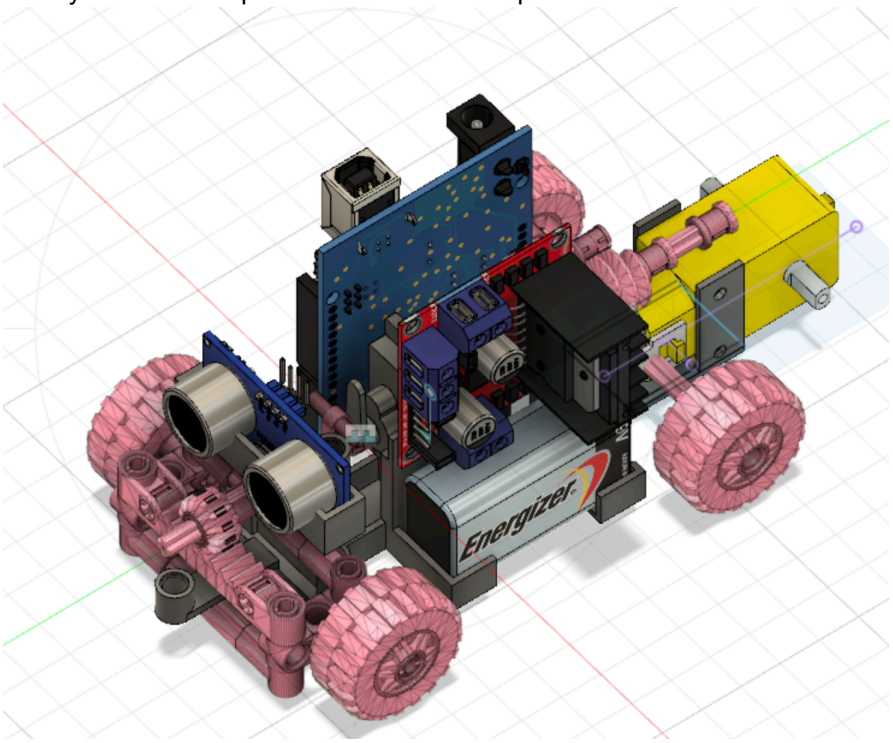
Designing and Building a miniature Car

The design

The car consists of a 3D printed PLA base which sustain the hole system : all batteries, the arduino, the servo, the motor driver and the motor and motor gear. This 3D design, is made using fusion 360, and takes into account all the rotating axles in its manufacure. Here is a picture of it :



Here you can find a picture of it with all components:

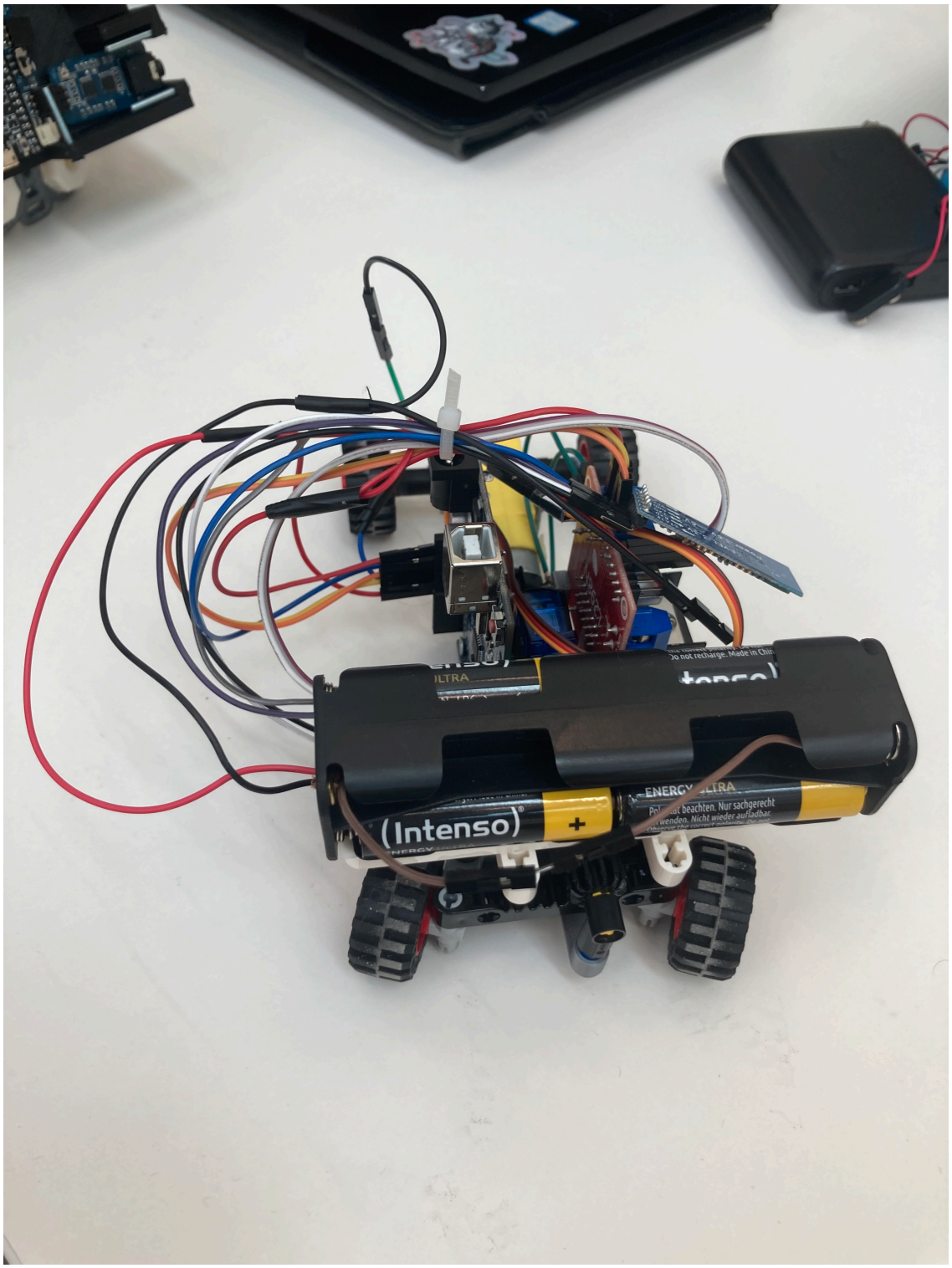


Material used

1. An Arduino Uno board
2. A L298N dual H-bridge motor driver
3. A small electric motor
4. A SG90 micro servo, black or transparent blue
5. Jumper cables male-male, male-female, and female-female
6. Two 9V battery
7. Two 9V battery connector
8. Six 1.5V battery
9. A 1.5V x 6 battery connector
10. A HC-05 bluetooth module
11. A motor gear
12. Rotating pieces of Lego technic from set 42116
13. 3D printed design

Features of the miniature Car

The miniature car, as you can see it below, uses the bluetooth module to communicate with the E3K board which is on the head set, A motor gear to move. As you can see below :



8

Conclusion

In this project, we demonstrated the feasibility of acquiring and processing three types of biosignals and controlling a LEGO vehicle using these: the vehicle's speed on the basis of the EEG signal, the steering of the left/right direction on the basis of the IMU signal, and the control of the backward/forward direction on the basis of the EMG signal. Further, we were able to 3d-print an adjustable headset in order to mount these sensors comfortably to the head and also explored the use of self-made electrodes and DIY-circuits.

While, in our project, all of these components were brought together for the sake of controlling a LEGO vehicle in an entertaining, game-like fashion, we believe that some of the components could also be applied for more practically useful settings in the future. What appears especially noteworthy here is the utility of the EEG signals: these differ from the other biosignals we recorded in that they do not just enable a convenient control of the car (like a remote control would, too) but allow for an extraction and visualization of something not evident through visual inspection (here, cognitive alertness).

We were initially very cautious about the expected low signal-to-noise level and what could feasibly be achieved using our relatively low-cost EEG setup, but we were positively surprised by what appeared to have, in fact, been possible. Beyond recreational purposes, we therefore believe that low-cost portable EEG systems capable of detecting brain waves at different frequencies could be applied to several domains and could provide project ideas for future generations of this course and beyond. For instance, thinking about the domain of education and work/life balance, students could use a portable EEG system to record their level of attention when studying, to automatically remind them of when best to take breaks, to avoid spending too many hours attempting to study without achieving much. In this setting, instead of linking the extracted EEG signals to a car, they could be linked to a mobile phone application that sends a notification once the focus dropped for a longer time and a break seems recommended. Alternatively, to avoid the usage of phones in study-related settings, the student could be notified through a to-be-engineered vibrating bracelet or similar tactile device. Further, when engaging in meditation or mindfulness exercises, or any other activity that may benefit from tracking longitudinal changes in one's ability to focus, a portable EEG system could potentially be used. Finally, an adapted version of such a system could possibly be used as a sleep stage tracker, allowing you to observe in what sleep stages you were for how long in the previous night. Similar to the interactive education setting mentioned above, such a tracker could even be linked to a phone or tactile device to wake you up when being in a light sleep stage within a well-defined time range.

While we acknowledge that many of the above-mentioned ideas and possible are likely to already exist in some form, we just wanted to express or enthusiasm for the opportunities emerging from mobile EEG and present some ideas for future generations of this course!

References

[1] *Do-it-yourself EEG guide by Elizbeth Ricker*. http://fab.cba.mit.edu/classes/863.15/section.Harvard/people/Ricker/htm/Final_Project.html. Accessed: 2022-06-03.

[2] *Original do-it-yourself EEG, EKG, and EMG guide*. https://sites.google.com/site/chipstein/homebrew-do-it-yourself-eeg-ekg-and-emg?authuser=0. Accessed: 2022-06-03.

[3] Paul L Nunez and Ramesh Srinivasan. "Electroencephalogram". In: *Scholarpedia* 2.2 (2007), p. 1348.

[4] Isabelle Constant and Nada Sabourdin. "The EEG signal: a window on the cortical brain activity". In: *Pediatric Anesthesia* 22.6 (2012), pp. 539–552.

[5] Marwan Nafea, Nurul Ashikin Abdul-Kadir, Fauzan Khairi Che Harun, et al. "Brainwave-controlled system for smart home applications". In: *2018 2nd International Conference on BioSignal Analysis, Processing and Systems (ICBAPS)*. IEEE. 2018, pp. 75–80.

[6] *Website of the EEG lab at the Princeton Neuroscience Institute*. https://pni.princeton.edu/research/pni-facilities/eeg-laboratory. Accessed: 2022-06-03.

[7] *Official Amazon link of the Mindflex game*. https://www.amazon.com/Mattel-P2639-Mindflex-Game/dp/B001UEUHCG. Accessed: 2022-06-03.

[8] *Wikipedia article about Polysomnography, with pictures and details of the typical setup*. https://en.wikipedia.org/wiki/Polysomnography. Accessed: 2022-06-03.

[9] *Polysomnography article, with detailed videos of the practical setup*. https://healthjade.net/polysomnography/. Accessed: 2022-06-03.

[10] *Brainvision Brain Cap figure*. https://brainvision.com/wp-content/uploads/2020/05/P_BrainCap_MR_semilateral.png. Accessed: 2022-06-03.

[11] *Wikipedia article about electroencephalography, with a visualization used as part of Figure 3.1*. https://en.wikipedia.org/wiki/Electroencephalography. Accessed: 2022-06-03.

[12] Sharda Shalikrao Kakde and Bashirahamad F Momin. "THE IMPLEMENTATION OF THE HUMAN COMPUTER INTERACTION SYSTEM BASED ON BIOELECTRICITY". In: *International Journal of Engineering Applied Sciences and Technology* 6.2 (2021), pp. 105–110.

[13] *Ultracortex Mark III Nova Revised*. https://github.com/OpenBCI/Ultracortex/tree/master/Mark_III_Nova_REVISED. Accessed: 2022-06-03.