# Flashing Jacket

Final Report

Alex Michon, Dinesh Parimi, Jack Wan

December 15, 2017

## Overview

Cyclists indicate their direction to drivers using arm signals. There are three different signals: left, right and stop (see Figure 1). Unfortunately, these signals may be hard to see at night and this can be dangerous.

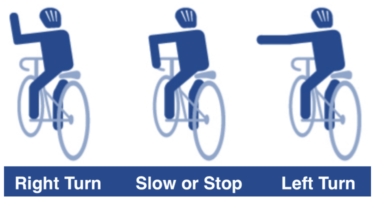

Figure 1: Arm signals

The goal of our project is to create a wearable LED grid to make biking safer at night. Our device will detect arm movements of the cyclist and create light signals to indicate in which direction the cyclist wants to turn.

We also thought about other potential applications of a wearable led grid, so we will implement another feature. Our device will be able to respond to ambient noise. For example, it should be able to display a spectrogram based on the current music.

## State Machine

In order to satisfy the requirements specified in the previous sections, we developed a state machine. We developed multiple classifiers to improve the performance. If there is a gesture detected, we will then detect if that is a switch signal, otherwise, the state machine will take the default transition directly to MAIN state. In the MAIN state, we use the switch signal to determine whether the state machine should switch between the BIKE mode and the MUSIC mode or not, which will further generate LED control signals to the LED matrix. Eventually, well output the light signals to the real physical world.

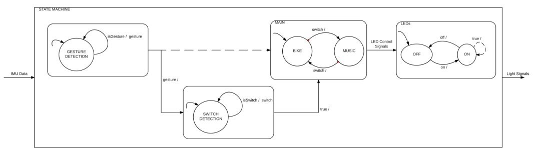

Figure 2: High level overview of the state machine

Within the BIKE state, we have the following state refinement, in which the raw data will finally be classified as a abstract gesture.

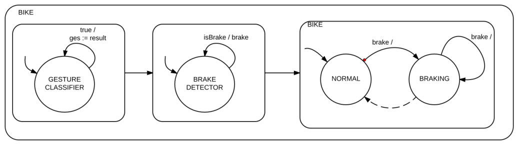

Figure 3: State Refinement of BIKE

Finally, the state machine will generate LED matrix control signals in both NORMAL state and BRAKING state. The color of all the signal generated in BRAKING state will turn into red, which is the only difference between NORMAL and BRAKING state. The following graph illustrates the NORMAL state:

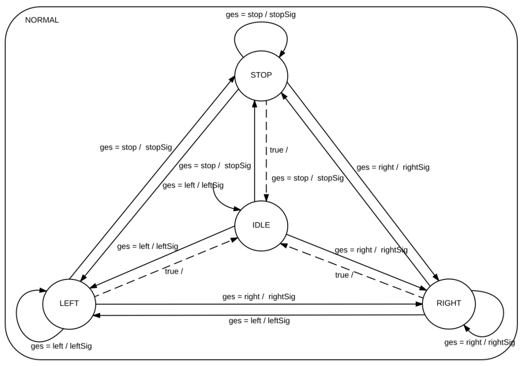

Figure 4: State refinement of NORMAL

## Bike gestures

### Sensing

In order to sense arm signals, we planned to use three IMUs: one on the forearm, one on the arm and one on the torso of the cyclist. Using these 3 IMUs, we would be able to classify each gesture. Plus, we would be able to use the acceleration and angular velocities from the torso IMU to get the values in the bike frame. This would have helped us to reduce the noise in our data due to the movement of the bike.

We used the Adafruit 9 DoF IMU LSM9DS1 and we managed to connect three of them to our microcontroller using SPI with the library provided by Adafruit [1]. The major drawback with SPI is that it requires a lot of wires: we needed 8 wires per IMU. We realized that 2 of these wires were used to get magnetometer values, which we dont need. So we modified the library to get rid of them.

However, we still needed 6 wires per IMU and we will be moving with them. For our prototype, there were contact failures due to the arm movement. We used jumper wires for most of our connections, which worked fine while testing separate components, but when we integrated, the number of wires grew exponentially. We had to chain jumpers together to make longer connections, and the inherent lack of reliability in jumper wires introduced many failure points.

We decided then to use only one IMU on the forearm. We attached the microcontroller very close to the IMU on the arm to reduce the probability of contact failure. With more time, we would have tried to solder the wires but this might have reduced the mobility of the arm.

### Classification

**Window:** Since the raw data is time-series data, we first set a fixed sized sliding window, and a fixed overlapping size. We fixed the sample rate to 10Hz. We then decided to use the IMU data from the last second to classify so the window size is 10, and the overlapping size is 9.

**Dimensionality Reduction:** In order to increase the performance of the classification, we implemented a dimensionality reduction algorithm. We compared the results of two algorithms: Linear Discriminant Analysis (LDA) and Principal Component Analysis (PCA). LDA provides better results since it is looking for the dimensions that maximum the variance between classes while PCA is looking for the dimensions that explains the most overall variance.

**Training the model:** We tested two types of classifiers: k nearest neighbors (kNN) and Gaussian Naives Bayes (GNB). In order to tune the kNN classifier, we plot the accuracy of it when k grows, and we finally found that the classifier performs the best when k is around 10. So we finally decided to set the k to be 13, after which we get the decision boundary of k-NN.

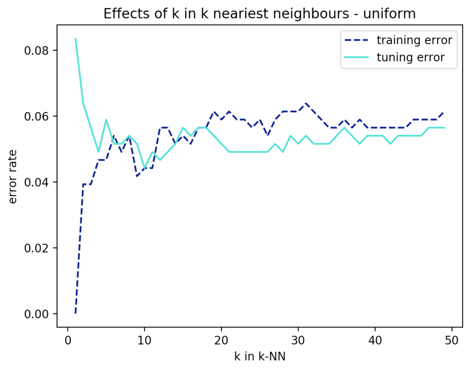

Figure 5: Error rate when k grows

As for the GNB, the decision boundary is very similar to k-NN except for that it is smoother than k-NN, which informs us that k-NN tends to over-fit the training data.

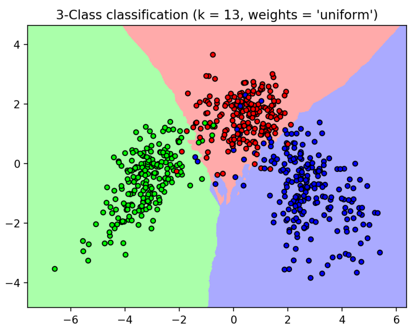

Figure 6: Decision boundary of kNN

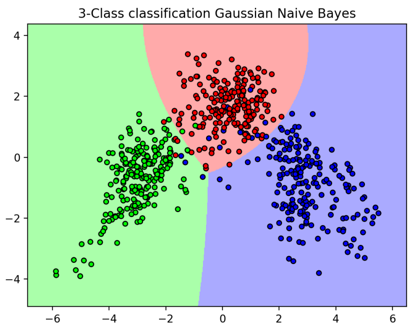

Figure 7: Decision boundary of GNB

**On the microcontroller:** The two classifiers give similar results so we decided to choose the classifier that would be easier to implement on the microcontroller.

- The classification algorithm using kNN requires building a kd-tree and including every training sample on the microcontroller. Building a kd-tree algorithm is not easy and the amount of flash memory is limited, which would in turn limit the number of training samples we could use. This was not a desirable outcome.

- GNB classification requires computing a weighted distance to the means of each class. Since we only have three classes, the amount of data to use on the microcontroller is small and we are not limited in our training set. The algorithm itself is also easier to implement.

So we decided to use the GNB classifier.

In order to execute the classification on the microcontroller, we referred to the scikit-learn library[4] written in Python and converted the LDA transformation and the GNB prediction functions into C++. We also wrote some tests to make sure that the classification was the same in C++ and in Python with the library.

## Music

For the music mode, we used the Adafruit I2S MEMS Microphone Breakout. This microphone is compatible with the ArduinoSound library developed by Arduino [2], which provides useful audio analysis tools like FFT. The main drawback is that it uses a lot of flash memory: 200kB, which represents 80% of the Feathers flash memory.

We used a sample rate of 8000 Hz in order to detect sound up to 4000 Hz. We then created a moving average on the magnitude of each frequency bins. Finally, we detected values that are above this average and converted the difference into a number of LEDs to light up in a column of our grid.

## Visualization

### LED grid

We built our LED grid using Adafruit NeoPixel LED strips. We chained ten strips of ten LEDs to create a 10x10 matrix. The LEDs are addressable using PWM. We then tested it using the Adafruit NeoPixel library [3] on an Arduino Uno.

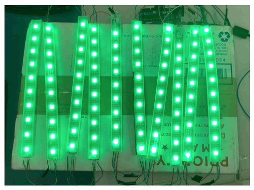

Figure 8: Idle signal

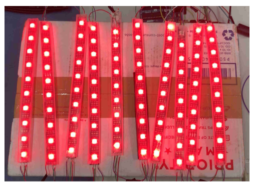

Figure 9: Brake signal

During the integration, we faced an issue: the generation of PWM output is not the same on the Arduino Uno and on the Adafruit Feather M0. Even if both are implemented in the library, it only worked with the Arduino and not with the Feather. Since we couldnt run our program on the Arduino due to

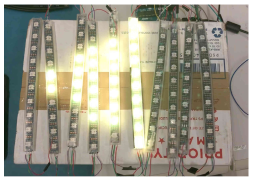
Figure 10: Left signal

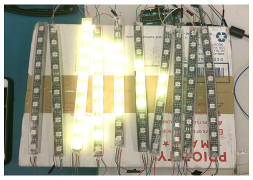
Figure 11: Right signal

a lack of flash memory, we decided to adopt use the two microcontrollers. We used the Feather and the Arduino connected over I2C - the Feather handling the main integrated software, and the Arduino being used as a slave device to receive commands from the Feather and control the LED grid.. We are aware that this is not an ideal solution. A better way to solve this problem would be to find a microcontroller compatible with the LED strips and with enough flash memory.

### Simulation

In parallel to the construction of the LED grid, we decided to implement a simulation of this visualization. We coded a small graphical user interface using Qt to create a virtual LED grid. Instead of producing PWM signals to the LED grid, the microcontroller would emit a string encoding the color value of each LED. The simulation would then parse this string and display the correct colors on the grid.

Because of our issues with the physical LED grid, the simulated grid was very useful for testing algorithms and sensors. We were able to see the results of our gesture classification and audio analysis before managing to connect the LED grid to the microcontroller.

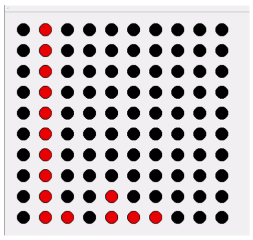
Figure 12: Screenshot of the simulation

## Results

Our final prototype is a fully-functional, stripped-down (with one IMU rather than three) version of what we initially proposed. We delivered an integrated software-hardware package consisting of an Arduino, a Adafruit Feather, a grid of NeoPixels, and a single IMU that could classify the different gestures (left, right, brake, and idle), as well as a software simulator that could be used with the Feather for both turn signaling and the music mode. After training our classifier model with updated data, we achieved successful gesture identification rates of 96%.

## Improvements

As said before, a major part to improve is the reliability of the connection between the microcontroller and the IMUs. Using three IMUs would make it possible to classify gestures on a bike. We would be able to subtract the acceleration of the torso accelerometers from the arm accelerometers readings to get values in the bike frame. We also didn't consider the possible noise in data collected on a bike. Collecting the training data from a real bike can be difficult, but we could use data augmentation to add noise the data artificially in future iterations. After data augmentation or bike data collection, we could then apply a low pass filter to the data to filter the noise in the data.

Another issue that we would need to solve is the control of the LED grid. We managed to workaround this problem by using two microcontrollers but we would need to find a better solution and use only one microcontroller.

## References

1. `https://github.com/adafruit/Adafruit_LSM9DS1`
2. `https://github.com/arduino-libraries/ArduinoSound`
3. `https://github.com/adafruit/Adafruit_NeoPixel`
4. `https://github.com/scikit-learn/scikit-learn`