POLITECNICO DI TORINO

Master's Degree in Software Engineering

Master's Degree Thesis

Analysis of commercial BCI solutions for automotive applications

Supervisor

Prof. Massimo VIOLANTE

Candidate

Nicola SABINO

July 2020

Abstract

Nowadays the influence of modern technologies has dramatically changed the concept of the vehicle in our community, moving from a simple means of transport to a complex machine able to meet driver needs.

The automotive division of the Luxoft company, situated in Torino (Italy), within its research and development branch, is studying different approaches and techniques to improve and extend the driving experience. All the efforts are concentrated on the Tele-Operated car, an electric vehicle that can be driven via a network interface. It accepts UDP packets as input messages and converts them to commands for electrical actuators situated on the steering column and the pedals of the car. In that context, the Xavier project is born.

The Xavier project, named like the fictional character with telepathic powers appearing in the X-men comic universe, is a long term development project for interns in Luxoft. This project presents the development process of an end-to-end infrastructure between driver and vehicle, thanks to the Brain-computer interface technology. The main objective has been to deepen the effectiveness of commercial BCI solutions such as an interface for drivers affected by disabilities. The project is reaching, with the end of this internship experience, the third development iteration. During these iterations, the whole know-how of BCI technology applied to automotive applications has been transferred and enhanced.

The starting point of this thesis was the first implementation of the Xavier application (version 1.1) implementing the base commands needed to achieve the first ride, and a series of tests made directly on the road about the reliability of the system.

During this thesis project, a new application (v2.0) has been developed looking for an overall improvement in terms of capability and usability. A new graphic user-interface has been provided and the new version is currently able to interoperate directly with the official application provided by the BCI manufacturer. Besides a complete, reliable, and safe simulation environment has been deployed to test the system indoor; the simulation environment has been taught to be extensible, easy-to-configure and to be as light as possible from a computational cost point of view.

Table of Contents

Acronyms

AI

Artificial intelligence

API

Application programming interface

BCI

brain-computer interface

CLI

command-line interface

CUDA

Compute Unified Device Architecture

EEG

Electroencephalography

FFT

Fast Fourier Transform

GPU

Graphics processing unit

MOC

Meta-object compiler

URDF

Unified Robot Description Format

VM
: Virtual machine

ROS
: Robot operating system

SDF
: Simulation Description Format

Chapter 1

State of the art

The study of electrical phenomena in the brain was conducted on animals as far back as 1875 when physician Richard Caton published his research from experiments on rabbits and monkeys in the British Medical Journal.

In the 1920s, a German scientist named Hans Berger was the first recording electrical signals produced by brain activity. During his research, he developed a technique to record those signals by positioning electrodes on the scalp of the patient: this method was called *Electroencephalography* - EEG.

In 1960s, psychologist Joe Kamiya with his experiments has shown the further capabilities of EEG technology. He demonstrated that those brain signals could explicitly be controlled by a human subject after a reasonable amount of training time using the feedback received by the EEG machine.

Thanks to those researches, in the '70s the term Brain-computer interface was coined referring to systems that take a bio-signal measured from a person and predict some abstract aspect of the subject's cognitive state.

Nowadays this technology is widely spread and accessible thanks to the advance in computational capabilities of modern calculators, the drop of in the cost in chips manufacturing, and their miniaturization. Besides, thanks to its diffusion, today we can find several types of BCI, with different types of electrodes, disparate processing procedures, and various levels of usability and reliability.

1.1 The Nervous system

The subject that analyzes and studies the nervous system is called neuroscience. The nervous system is in charge to coordinate actions and collect sensory information by transmitting signals to and from different parts of the body. It is composed of two main parts, the central and peripheral nervous system: the first includes the brain and spinal cord while the second consists mainly of nerves - cords made of fibers.

1.1.1 The Brain

The brain, the main organ of the nervous system, can be divided into different areas, each one with its purpose:

- Occipital lobe: recognize objects and vision.
- Temporal lobe: visual memory, language abilities, and emotion association.
- Frontal lobe: emotions, reasoning, planning, problem-solving, judgment, movement, and parts of speech.
- Cerebral Cortex: thinking, voluntary movements, language, reasoning, and perception.
- Cerebellum: movement, balance, posture, and coordination.
- Hypothalamus: body temperature, emotions, hunger, thirst, appetite, digestion, and sleep.
- Thalamus: acts sensory and motor integration.
- Pituitary gland: hormones production
- Pineal gland: control growth.
- Amygdala: emotions.
- Hippocampus: learning and memory.
- Mid-brain: breathing, reflexes, and swallowing reflexes. (Includes the Thalamus, Hippocampus, and Amygdala)
- Pons: motor control and sensory analysis.
- Medulla Oblongata: maintains vital body functions, breathing, digestion, and heartbeat.

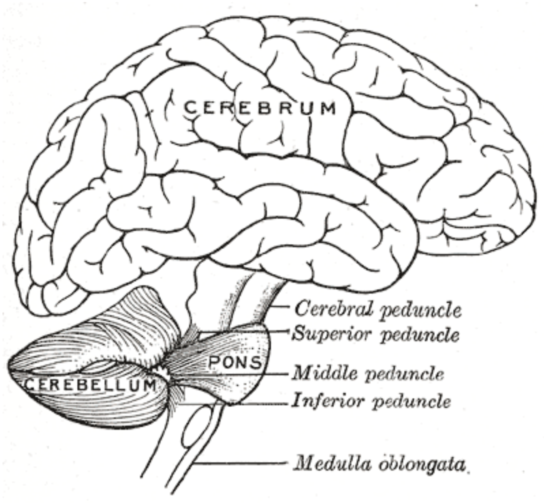


Figure 1.1: The brain - [Gray's Anatomy 4, H.V. Carter's illustration]

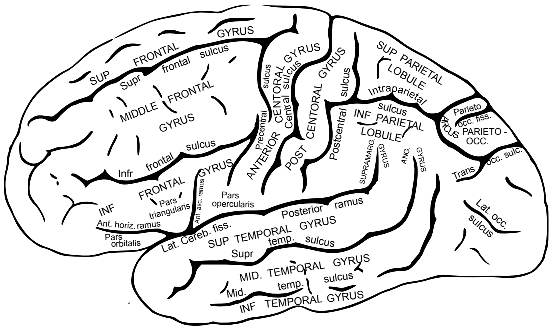


Figure 1.2: Detailed representation of the cerebrum - [Gray's Anatomy 4, H.V. Carter's illustration]

1.1.2 Neurons

The nervous system is characterized by a special type of cell, called *neuron.* Neurons can receive integrate and forward electrical and chemical signals. The electrical signals are caused by a rapid, temporary change in membrane electrical charge while the chemical signals - defined as neurotransmitters - are released from one neuron as a result of the electrical signal.

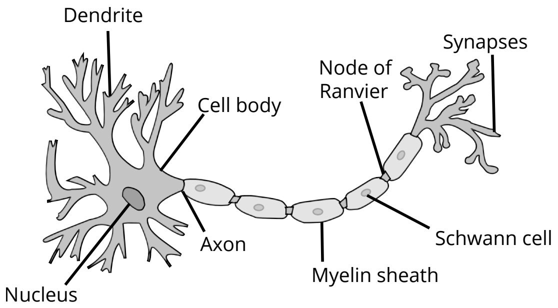


Figure 1.3: The Neuron cell -"Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology, and End Results (SEER) Program.

Neurons are composed by:

- The Soma, cell body containing the nucleus
- Dendrites, a structure composed of branches able to receive neurotransmitters from other neurons.
- Synapses, end junction of a neuron connected with the dendrites of another neuron.
- Axon hillock, able to forward signals emitted by the nucleus of the neuron, it "merges" signals coming from multiple synapses
- Axon, a structure that propagates the signal. It carries the generated potential - electrical signal - to the next neuron. A Neuron could have one or two axons. Some axons are covered with an insulator material called *myelin* to minimize the dissipation of the electrical signal.

1.2 Neuroscience tools

Nowadays the study of the nervous system can rely on a large collection of instruments exploiting different approaches along with different physical phenomena related to brain activities. In this section, we will deep into EEG technology as the main technique and its support to a Brain-computer interface.

1.2.1 Electroencephalography

The electroencephalography procedure allows us to record and evaluate the small electrical activity on the scalp on the subject generated in the outer part of the brain, the cerebral cortex.

The EEG procedure can help doctors in medical diagnosis comparing the collected data against recognizable wave forms, researchers to study human behavior, and could helps people to improve their productivity and wellness. This safe and painless technique is also helpful for diagnosing or treating some disorders like Brain cancer, damages from injury, sleep disorder, and so on.

1.2.2 Brain waves processing

The neurons emit and receive ionic currents causing bio-potentials registered by electrodes up to several thousand times per second. EEG signals are then amplified, digitized, and sent to a calculator for data processing.

There are different types of data processing available: the most common is to analyze all the acquired signals and divide them into bandwidths to describe their functions. Most sophisticated techniques involve Machine Learning techniques and classification algorithms with the purpose of properly distinguish and catalog complex waveform without the judgment of a human being. These require a lot of ground data to train and tune properly the algorithms.

Brainwaves are commonly classified by frequency into five main categories: β Beta, α Alpha, θ Theta, δ Delta, and γ Gamma.

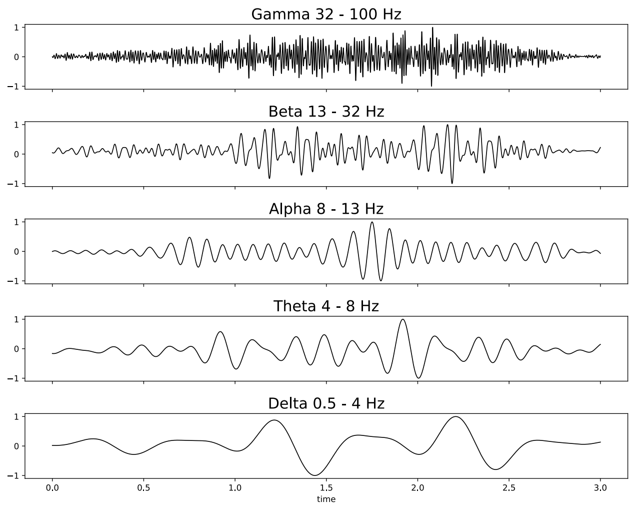


Figure 1.4: Brain waves with different frequencies.

- δ waves (0.5 - 4 Hz) arise during meditation, in a state of deep sleep or coma.
- θ waves (4 - 8 Hz) appear in sleeping or daydreaming.
- α waves (8 - 13 Hz) can be induced by closing the eyes and relaxing, and they are rarely present during intense cognitive processes like thinking, mental calculus, and problem-solving.
- β waves (13 - 32 Hz) characterize a conscious and alert state. These frequencies are particularly distinguishable during logical-analytical reasoning.
- γ waves (32 - 100 Hz) are related to learning, memory, and information processing.

1.2.3 EEG devices

We can split EEG machines into two main categories: clinical and consumer devices. Clinical devices impose participants to do not move while data collection , and the monitoring has to be done in a controlled environment to avoid distorting the signal. On the other end, consumer devices allow us to monitor brain activity in movement, with a certain degree of freedom.

Furthermore, EEG devices could be wearable caps, usually preferred in clinical applications since they support more sensors due to the larger surface area for electrode placement; rigid headsets whit fewer electrodes, usually used in consumer applications.

EEG devices support different types of electrodes. The most common are wet or dry reusable electrodes. Wet electrodes ensure finer data accuracy since they use an adhesive gel for better contact with the scalp of the patient: they are usually used in caps devices for clinical solutions. Dry electrodes don't require an adhesive gel, these electrodes are often used in consumer applications since they allow quicker setup time.

1.2.4 10-20 system

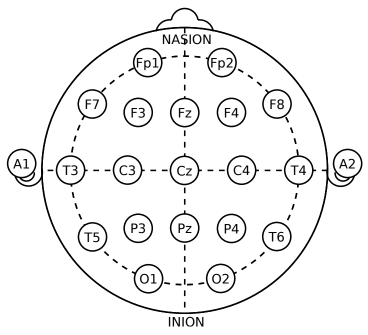


Figure 1.5: Electrode locations of the International 10-20 system for EEG (electroencephalography) recording - wikimedia.org

The 10–20 system is an internationally recognized method to define the position of the electrode on the scalp of the subject. This method was developed to ensure the standardization of exams, avoiding external biases in different tests according to the scientific method.

The system is based on the relationship between the location of an electrode and the covered cerebral cortex section of the brain. Each electrode has a letter to identify the covered brain area: pre-frontal Fp, frontal F, temporal T, parietal P, occipital O, and central C. The letter Z - zero - refers to electrodes placed on the mid-line plane of the scalp while the A refers to the prominent bone process usually found just behind the outer ear. Each electrode is also associated with a number, even for the right-hand side of the head and odd for the left-hand side.

The method takes its name to the fact that the actual distances between adjacent electrodes are either 10% or 20% of the total front-back or right-left distance of the skull.

1.2.5 Advantages and Disadvantages

EEG offers two main advantages respect other brain measurement techniques.

The former is high precision time measurements. Changes in the brain's electrical activity occur very quickly, and extremely high time resolution is necessary to detect accurately bio-potentials.

The latter, unlike other electrical recording devices that require inserting electrodes into the brain, EEG electrodes are simply stuck onto the scalp of the subject. It is a non-invasive procedure that allows researchers to efficiently access a human brain without surgery.

Moreover, EEG equipment is more affordable respect other devices, simpler to operate for the medical staff or researchers, easy to maintain, and also portable and versatile for the ones with wireless capabilities.

On the other hand, the main disadvantage of EEG is the weak spatial resolution, due to the usage of superficial electrodes on the scalp of the patient. The perceived signal of a single electrode is the result of chemical-electrical interaction between hundred of thousand neurons in one squared centimeter of the brain cortex. Besides, particularly strong signals can be collected by several neighboring electrodes. Because of this, EEG cannot distinguish neatly specific signals coming from adjacent locations of the brain cortex.

1.2.6 EEG and Brain-computer interface

EEG technology is capable to extract brain signals from a subject with a certain degree of spatial and temporal accuracy. A Brain-computer interface collects these signals and uses machine learning algorithms to translate these informations into valuable and understandable commands for external actions. The machine learning algorithms are trained to detect emotions, actions, and expressions by EEG signals. When the algorithms match a trained command it runs a defined action like spin a motor, turns on a relay, moves a cursor, and so on. The most common application of this solution are usually Spellers, wheel-chair driving, control of synthetic body joints, advanced in-game interaction, and so forth.

Chapter 2

Hardware Equipment

In this chapter, we will introduce the hardware equipment involved in our project. The goal is to build a complete BCI interface between a driver and a car by introducing as few components as possible, promoting ease of use, reliability, and the comfort of the operator. The equipment chose has been performed preferring commercial solutions instead of custom-built ones to increase the degree of security as the task to be performed includes risks of injury for the driver.

2.1 Headset

The headset chosen for this project is the Emotiv epoc+, a wireless fourteen channels device. It is a commercial solution composed of the headset itself and some software tools for different activities like raw EEG data recording as well as advanced data processing. Headset connectivity relies on Bluetooth low energy technology and a proprietary USB receiver running at 2.4GHz.

The device is characterized by reusable, wet electrodes made of felt and soaked with a common saline solution available in supermarkets and pharmacies. The electrodes are arranged following the 10-20 standard in the following positions: AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4. The device provides 128 or 256 samples per second with a resolution of 0.51µV and a bandwidth of 0.16 - 43 Hz. Moreover, the headset is equipped with an Inertial measurement unit whit accelerometer, magnetometer, and gyroscope.

The Emotiv company sells this product with different licensing options for developers: free subscription plan offers limited capabilities of the system, researcher subscription plan allows advanced features while enterprise subscription offers custom configuration and more flexibility.

In this project, we adopted the free subscription plan for a set of Application programming interface (API) using secure web socket connection which offers:

- Mental commands: the possibility to train and detect thoughts associated with abstract concepts like left, right, up, down and so on.

- Performance metrics: 6 features that describe the cognitive state of the subject. The available states are engagement, excitement, stress, relaxation, interest, and focus. These metrics are available with a sampling frequency of 0.1Hz

- Frequency Bands: gives information on 4 brain waves β, α, θ, and γ.

- Facial expressions: the headset is capable to catch some facial expressions like frown, smirk, smile, wink, and so forth.

- Motion data: gives information about the movement of the head.

The whole thesis project refers to the second version of the API. This version is no longer compatible with older ones and offers a different syntax and new features.

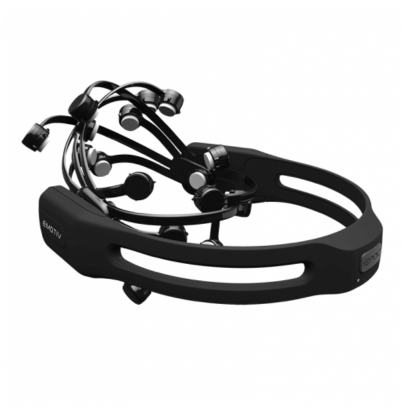


Figure 2.1: Emotiv EPOC+ 14 Channel Mobile headset

Emotiv products hold CE FCC and UL certification marks, accomplish with all international product safety standards including radio frequency emissions and electrical safety, as well as other issues such as possibly toxicity or allergic reactions to components.

This headset has been chosen for its reliability, construction materials, ease of use, safety and, BCI capabilities. It is also an affordable product for an end customer in a perspective of a final profitable solution, as an optional for a car.

2.2 Paravan system

Paravan Gmbh company offers mobility solutions and automotive products for people with different disabilities. The solutions offered include car conversion, lifting and loading systems, custom security systems, and drive-by-wire integration.

The drive-by-wire system consists of an additional car control unit able to communicate via bus with the main one and collect custom messages encoded in a proprietary protocol. It is able to control stepper and linear electric-motors installed on the steering column and the pedals. The product is certified ISO 16750: Road vehicles — Environmental conditions and electrical testing for electrical and electronic equipment.

In this thesis project the Paravan system is installed in a front-wheel-drive full electric car; BMW i3, model-year 2013 with 43,5 horsepower.

2.3 Additional virtual cockpit

The final user interacts whit the whole system through an additional cockpit mounted into the car dashboard. The cockpit is composed of a fourteen inches display and a windows personal computer with 8 GB of RAM, an Intel i7 processor, and 275 GB of disk space; running all the required software for the Emotiv headset and a further custom program explained more in details in the next chapter.

Figure 2.2: The teleoperated car

Chapter 3

Application development

The core section of this project is the realization of a comprehensive application able to interact with the wireless headset, communicate efficiently with the end-user, and forward all the collected information and commands to the Paravan system.

In the following chapter we will dive into the selected programming languages and the Qt framework technology, which characterize the core application, called Xavier.

The Xavier application is designed to offer a full and integrated experience of the whole system, offering the possibility to configure, train, and run the headset system and control the car with it.

We will deeply analyze the program structure, the design choices, and the integration with the two external systems: Paravan and Emotiv.

Moreover, we will analyze in detail the features offered by the Emotiv Cortex v2 API available with a free licensing plan.

3.1 Qt framework

Qt is a cross-platform application development framework written in C++ for desktop, embedded, and mobile devices. In a Qt project, before the compilation step, a preprocessor called MOC - Meta-Object Compiler - parses the source files written in Qt-extended C++ and generates standard-compliant C++ sources. In this way, all generated sources from the preprocessor can be compiled by any standard C++ compiler like Clang, GCC, ICC, MinGW, and MSVC. Qt adopts qmake as a default cross-platform build system, but it is also compatible with CMake and others.

The Qt framework represents currently one of the leading technologies in resource-aware applications, adopted as a reference in many commercial products.

3.1.1 Qt messages system

One of the significant improvements with the adoption of the Qt framework is the inter-exchange of messages.

Each object in the Qt should extent the class QObject to inherit the capabilities and core functions offered by the framework. Those objects acquire the possibility to exchange messages via so-called slots and signals.

A signal is a public access functions emitted when a particular event occurs while a slot is a normal C++ function that is called in response to a particular signal. Qt offers several predefined slots for its QtObjects but it is also possible to define custom slots and connect them to a signal via the `connect()` method.

This mechanism is also type-safe, supports customize-able parameters, ensures correct timing, and relies on a loose connection: Objects which emit a signal don't care about which slots will receive it.

3.1.2 QtQuick and QML

The QtQuiclk module and the QML language are the latest instruments offered by the Qt suit in support of human-machine-interfaces and user experience design. These technologies establish different and intuitive approach respect to the older approaches, meeting the needs of developers and designers. Using the QtQuick module, those professional figures can easily build fluid animated user interfaces in QML and have the option of connecting them to any back-end C++ libraries.

QML is a highly readable declarative language that was designed to describe graphical user interfaces - GUI. Each GUI is described in terms of smaller elements, combined into a tree structure of complex components. It supports several animations, built-in control commands for transitions, and can be extended with JavaScript code.

When running QML, it is executed in a run-time environment implemented in C++. It consists of an engine, responsible for the execution of QML files, holding the properties accessible for each element or structured component.

Currently, this technology is strongly adopted in the realization of digital clusters and cockpits in the automotive industry.

3.1.3 Sub-directories system

A good practice for larger software projects is to isolate its components into different business-logic units: GUI files, logic, data persistence, and so on. The software components belonging to the same unit are usually combined into software libraries: `.lib` or `.dll` files in Windows, `.a` and `.so` files in Linux and `.a` or `.dylib` files in MacOSx.

The Qt framework using the qmake built system can arrange those sub-directories and define the correct pre-processing and compilation order. Each sub-directory of this mechanism is a complete and autonomous Qt project with its qmake file and configurations. The developer in this way is able to develop debug and test single software modules separately and link them statically or dynamically.

3.2 Cortex v2 API

Cortex API, built on JSON and WebSocket, allows developers to built third-party applications able to interact with the Emotiv BCI system. JavaScript Object Notation - JSON - is a standard interchange format that uses text to serialize and transmit data objects structured in key-value pairs and array data types. Respect other language-independent data format like XML is more compact and supports naively arrays, numbers, strings, and nested objects.

The WebSocket is a two-way full-duplex communication protocol layered over TCP. According to the IETF definition in the RFC 6455, the goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that do not rely on opening multiple HTTP connections.

To access to this instrument, a developer has to be registered to the Emoitv site, apply for one of the available plans, register the application and obtain a Client Id and a Client secret tokes that have to be used at connection time by the application. Once properly installed, On Windows and Mac PCs, the Cortex service is a background process that communicates with the headset and the Emotiv cloud and acts as WebSocket server on the port 6868. Furthermore, the Emotiv suite offers two utility applications: Emotiv App and Emotiv BCI. Emotiv App is a service application able to manage headsets and third-party apps permissions while Emotiv BCI is the official Emotiv application that let the users or developers to experiment with the headset.

3.2.1 JSON-RPC

Cortex server adopts a strict syntax in JSON messages called JSON-RPC which is a stateless, light-weight remote procedure call protocol. Each request object sent by a client must contain those fields:

- `jsonrpc` field containing the version of the protocol supported by the client.
- `method` specifying the name of the method to be invoked.
- `params` structure containing the parameter to be used during the invocation of the method.
- `id` a String, Number, or NULL established by the Client. If a request does not contain an id it is treated as a Notification and as such, no Response object needs to be returned to the client.

For each received request messages, with the only exception of Notifications, the server has to reply complying with the following structure:

- `jsonrpc` field containing the version of the protocol supported by the server.
- a `result` or `error` field containing a structured object.
- the same `id` as the value of the id member in the request object.

The following example shows the call of remote procedure `subtract` with its parameters and the response by the server.

A request message from a client

```
{
    "jsonrpc": "2.0",
    "method": "subtract",
    "params": {
        "minuend": 42,
        "subtrahend": 23
    },
    "id": 3
}
```

Response message from the server

```
{
    "jsonrpc": "2.0",
    "result": 19,
    "id": 3
}
```

3.2.2 Authorization and headset pairing

The first part of the authentication procedure requires that a registered user is logged in the Emotiv App application.

First, the method `getUserLogin` has to be performed to check if the user has already logged in though Emotiv App. Then calling the procedure `requestAccess` user is required to approve the third-party application directly in the Emotiv App. If the user has already approved your application, then this API will prompt nothing.

Finally, the call `authorize` generate a cortex token able to authenticate all the further request for the server. Once properly authenticated a client can query the API for available headsets with the `queryHeadset` procedure. From this point forward a program can establish a connection with the available headsets.

3.2.3 Session and data persistence

Each connection with a headset is based on a Session object. When a user needs to interact with the headset the application has to create a session first. Each application is allowed to open a session with only one headset at a time but can open several sessions with different headsets. Each session is opened and closed with the methods `createSession` and `updateSession`. A session is implicitly bonded to an application and is automatically closed when the application is disconnected from the Cortex service or the headset is disconnected.

A session is a temporary in-memory object and after its closure, it is destroyed by Cortex API. Changes of a session can be preserved calling the method `setupProfile` specifying the field status to save. To obtain information about the stored data associated with a profile is possible to call the method `getTrainedSignatureActions`.

3.2.4 Commands

Each profile is associated with a series of Training objects of two groups: fourteen mental command labels and twelve facial expressions.

The available mental commands are:

- neutral
- push
- pull
- lift
- drop
- left
- right
- rotateLeft
- rotateRight
- rotateClockwise
- rotateCounterClockwise
- rotateForwards
- rotateReverse
- disappear

While using mental commands the user is required to choose up to four labels, neutral included, and associate to them an abstract concept. At first, the user is required to train the neutral command to recognize the background mental state, then the user can train a new mental label.

It is strongly suggested by the Emotiv team to master one action and have good control before adding a second action and so on. The user has to be focused for eight seconds during the training and has to be relaxed so the algorithms don't have to cope with variable muscle signals. Train sessions can be varying from user to user, but to master a single command is suggested to train it at least for twenty minutes. It is up to the user what abstract concepts to associate to a mental command label: usually, more experienced users tend to visualize simple objects movement like moving a ball left, right, up and down.

The Emotiv headset has eight electrodes positioned around the frontal and prefrontal lobes which acquire signals from facial muscles and the eyes. The Emotiv detection system uses these signals to classify which muscle groups are causing them and use efficient classifiers to detect many facial expressions. The available facial expressions are dived into two groups:

Trainable commands

- neutral
- surprise
- frown
- smile
- clench

Non-trainable commands

- blink
- winkL
- winkR
- horiEye
- laugh
- smirkLeft
- smirkRight

Facial expressions are always detectable, even if they have not been trained and also in this case the neutral has to be trained at first.

3.2.5 Training procedure

API lets a client to register to different streams with the remote procedure `subscribe`. Once registered the client will receive periodical Notifications, messages without id field, by the server. The Cortex system offers different streams with different purposes:

- `eeg` - Raw EEG data, available only with paid subscription plans.
- `mot` - motion data from the headset.
- `dev` - Device data information like the battery level, the wireless signal strength, and contact quality of EEG sensors.
- `pow` - The alpha, low beta, high beta, gamma, and theta brain signals bands.
- `met` - Performance metrics detection.
- `com` - Mental commands detection.
- `fac` - Facial expressions detection.
- `sys` - The system events. These events are related to the training of mental commands and facial expressions.

At first, the client must subscribe to the data stream `sys` to receive the training events from Cortex. Then it has to perform the following steps:

1. Start the training by calling `training` procedures specifying the action to train and the control field `start`.
2. On the `sys` stream, the client receives the event `started`.
3. After eight seconds, it receives one of these two events:
   - The Event `succeeded`, the training is a success. The client can accept or reject it.
   - The Event `failed`, the data collected during the training is of poor quality, the client must restart the training
4. Call `training` with the control `accept` to add the training to the profile. Or using the control `reject` to dismiss this training.
5. Cortex API sends back the event `completed` to confirm that the training was successfully completed.

Clients must save the profile to preserve new training otherwise it will be lost unloading the profile. Furthermore, after the second and third step, it is even possible to send the control `reset` to cancel the training.

In the end, the client can conclude a training session unsubscribing from the `sys` stream with `unsubscribe` remote procedure.

3.2.6 Commands detection

The API let the possibility to register to several streams. Once properly subscribed to a stream that is different from the `sys` one, the client receives messages with regular frequency without and id field, Notifications in the JSON-RPC glossary, defined as events in the Cortex documentation. For our purposes, we focus on `fac` `com` and `met` streams.

The API classify facial commands in three main classes: eye movements, upper face, and lower face actions. Upper and lower face actions are accompanied by the upper and lower power fields, specifying the intensity of the command with a floating number over a range of 0 to 1.

```
["eyeAct","uAct","uPow","lAct","lPow"]
```

The facial stream provide events with 32 Hz in the following format

```
{
    "fac":["neutral","neutral",0,"clench",0.0576],
    "sid":"a4f69c56-9769-4a4d-950c-490eb5ebe372",
    "time":1559903035.2961
}
```

The mental stream presents data with a frequency of 8 Hz in the form of `["act","pow"]` where act is the label of the detected action and pow is the relative power expressed as a floating number over a range of 0 to 1.

```
{
    "com":["pull",0.564],
    "sid":"79cc669b-af2e-465a-bdc2-0e9bd4aebe80",
    "time":1559903099.348
}
```

Each performance metric is a decimal number between 0 and 1. For each metric, with the exclusion of excitement, there is also a flag `isActive` set to true if the detection is running properly. It is set false if the detection cannot be performed, for instance, due to a lack of EEG signal.

```
    ["eng.isActive","eng","exc.isActive","exc","lex",
    "str.isActive","str","rel.isActive","rel","int.isActive",
    "int","foc.isActive","foc"]
{
    "met":[false,null,false,null,null,false,
        null,true,0.266589,false,null,true,0.098421],
    "sid":"6a68b92a-cb1f-4062-bf1f-74424fbae065",
    "time":1559903137.1741
}
```

3.3 API mocking

Create mock-ups is a consolidated modern practice that improves the development process within complex projects. Mocking can be done at many levels; code, API, service, and can be used for multiple scenarios. Specifically, API mocking can support both development and testing operations.

To speed up the development of the Xavier application, a mocked version of Cortex v2 API has been developed to allow code testing in the absence of the real headset.

3.3.1 Node.js

Node.js is an open-source development platform, useful for building highly scalable and fast JavaScript applications. Node.js is built on v8, the JavaScript run-time engine that powers the Google Chrome browser. It is designed to be used in intensive asynchronous I/O applications, utilizing the non-blocking event-driven architecture. Each Node.js application runs in a single-thread environment and it can collect incoming requests in an event-queue. The main drawback of this design architecture is the fact that the system is poorly efficient in CPU-intense applications.

Node.js and its programming language JavaScript, together represent one of the most diffused and popular solutions in the market of web technologies, counting a very solid base community and a wide ecosystem of powerful modules.

3.3.2 Mocked Cortex v2

Mocked Cortex v2 is a Node.js application composed of a single JavaScript file. It is capable to emulate only a subset of the whole procedures available in Cortex v2; it supports login operations, training, and streaming remote procedures.

The Mocked application relies on the `WebSocket` node packet, provides a server service on port 7000, and is composed of a main call-back function a series of JavaScript functions, each one for a specific remote procedure. Each login and training procedure sends back to the client a predefined well-formed JSON response while streams subscriptions emulate a series of commands with random strength intensity.

Facial expressions and Mental commands subscriptions are stored in global variables using the `setInterval()` method and are released when clients request unsubscriptions with `clearInterval()`.

3.4 Xavier

Xavier is an exhaustive Qt application that is in charge to directly interact with the end-user, collect data from the headset, and send commands to the vehicle.

It has been developed to be fully functional with Cortex v2 API as well as the programs offered by the Emotiv company. The user can operate both from the Xavier program or using official Emotiv programs, loading saving and using the same profile; in this way, the user can do intensive training whenever he wants and uses the same training-profile while driving the car within our application.

The final goal was to provide a fully functional instrument to the user directly embedded in the car cockpit, taking care about ease of use and reliability exploiting the capabilities offered by the headset.

```
Xavier_v2
 ├─ Xavier_v2.pro
 └─ src
     ├─ gui
     │   ├─ gui.pro
     │   ├─ qtquickcontrols2.conf
     │   ├─ ...
     │   └─ qml.qrc
     ├─ cortexclient
     │   ├─ cortexclient.pro
     │   └─ ...
     ├─ businessLogic
     │   └─ businessLogic.pro
     │       └─ ...
     └─ qmsgpack
         ├─ qmsgpack.pro
         └─ ...
```

The whole Xavier project is divided into three independent sub-modules, each one with its resources and a qmake file. Furthermore, a master qmake file has been provided to establish the correct built order at compile time.

The `gui` sub-module is composed of a `.conf` file that specifies the look-and-feel appearance, a series of qml scripts, and `.qrc` qml resource file that list all the available scripts.

The `cortexclient` sub-module is a customized version of an official C++ the library offered by the Emotiv company page.

The sub-module named `businessLogic` is the supporting column of the whole program, composed of key components able to coordinate the interaction between gui and cortexClient.

In the end, we have the `qmsgpack` submodule that is used to convert lightweight text messages to a from Paravan system.

Besides, the sub-directories cortexclient, businessLogic, and qmsgpack has been treated as static libraries to speed up the compile time while the gui module contains the entry point of the program.

3.4.1 Human-machine interface

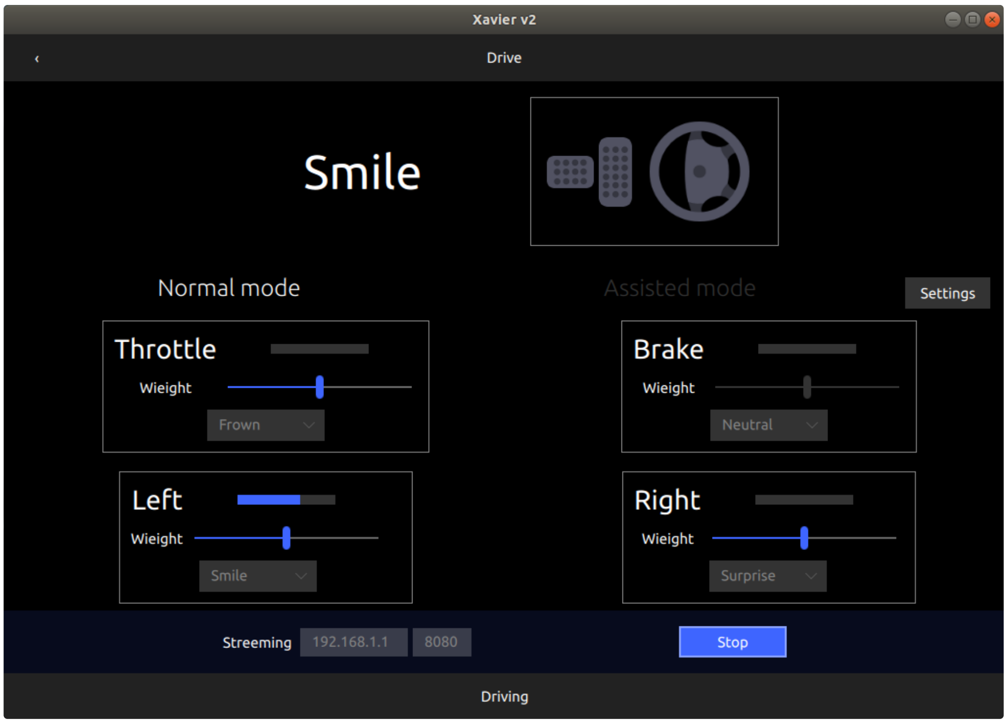


Figure 3.1: Xavier graphic user interface - drive page during a driving session

The whole GUI adopts the Microsoft Universal Design Guidelines with a dark theme to highlight the virtual elements and ensure good levels of contrast and visibility in every condition. The overall effect is minimal and essential to keep the user focused on the road.

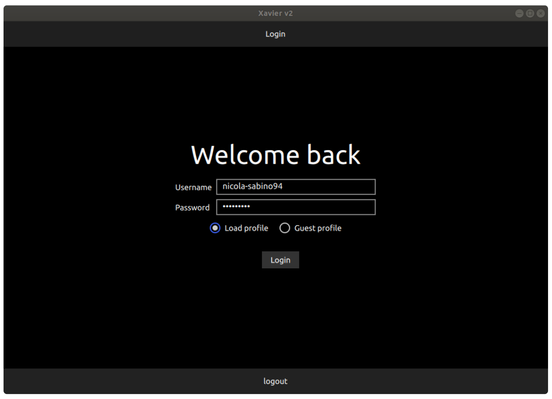
The view is structured in a mainframe with two tool-bars; the one in the upper part used for navigation in the application while the one at the bottom is used as a prompt to notify the user for events, errors, and so on. The two tool-bars surround the content composed by three overlapped pages: login page, training page, and drive page.

At first, the login page is displayed to require the user's credentials. During the login procedure, the app is capable to notify errors like no headsets detected by the system, wrong credentials, missing permissions in the bottom bar.

The training page lets the user select between two training sets: mental commands and facial expressions. The user has to train neutral state at first in both modalities, after that can select new commands to train or remove them, enforce their response with further training sessions and also reset their history. Each train session is supported by essential textual instructions and visual elements. The page also drives the user through the correct usage of the application locking components during training or when the neutral state has not been trained first.

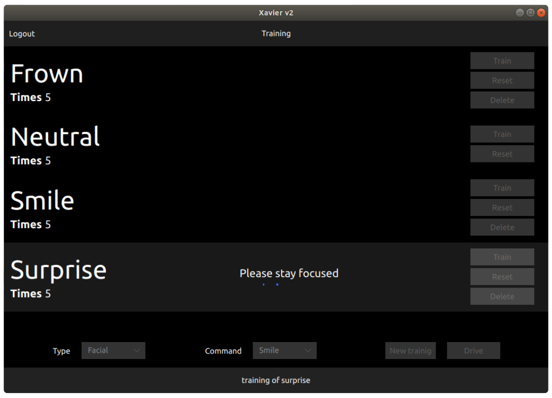
Once the user is satisfied with his training sets it can move on the next page with the drive button. The drive page is divided into three parts. In the center part the user can bind trained commands with the four car operations: throttle, brake, steer left, and steer right. For each car action, the UI offers a card layout showing the command, a process bar of the detected intensity of the command, a slider that controls the sensibility of the command, and a combo-box that gives the possibility to choose the command. In the bottom part, there is a connection box that let the possibility to specify the IPv4 address of the Paravan system and the stream button that starts sending the flow of detected commands to the car. In the upper part, the user has visual feedback of the detected commands by the Cortex system and the corresponding operation on the steering wheel and pedals of the car.

The entire application exploits the capability of qml language to natively handle animations and use them to increase the feedback with the user and keep it aware of critical operations. For instance, when the user presses the start button the whole connection box starts to glow regularly with the accent color while the steering wheel and the pedals move accordingly to empathized the fact that the commands will be sent to the car.



Username: nicola-sabino94
Password: •••••••••
[✓ Load profile] Guest profile

Figure 3.2: Xavier application - login page



Frown
Times 5

Neutral
Times 5

Smile
Times 5

Surprise
Times 5

Please stay focused

Type: Facial
Command: Smile

training of surprise

Figure 3.3: Xavier application - training of the Surprise facial expression

3.4.2 Back-end communication

The application is structured in a way to obtain a neat division and independence within modules. For this purpose, the cortexclient sub-module has to be maintained as much consistent with the official release of this component while the businessLoic sub-module coordinates the application behavior.

Cortexclient main characters are:

- `cortexClient` class, in chargeof establishing a web-socket connection and exchange messages or events with Cortex API service.
- `headset` class, representing a physical Emotiv headset.
- `headsetFinder` utility class, able to search connected headsets.
- `sessionCreator` class that creates a logical connection between the Emotiv headset and the Xavier application

The application logic relies on a `MainController` class that has to create a secondary thread to handle asynchronously API events from the ones fired by the HMI. Qt framework offers different solutions for multi-threading implementations; the selected one is the GUI thread - worker thread one. It consist to keep all the classes devoted to interact with the graphical user interface in the main thread and to wrap all the other operations in a utility class that will be inflated in a secondary thread: `binder` class in our case.

The binder is the common interface to all the functionalities offered by the cortexclient sub-module and it is connected to MainController with several qt signals. The MainController holds an instance of QThread via a `QScopedPointer`, a wrap in the qt framework of `std::uniqueptr` smart pointer that ensures a safe release of resources.

The QThread class offers predefined interfaces in terms of signals and slots to handle its correct life cycle; the nested Class can be linked to those signals to start its job and release gracefully its resources when the thread has to be closed. The complete thread creation is shown in figure 3.4.

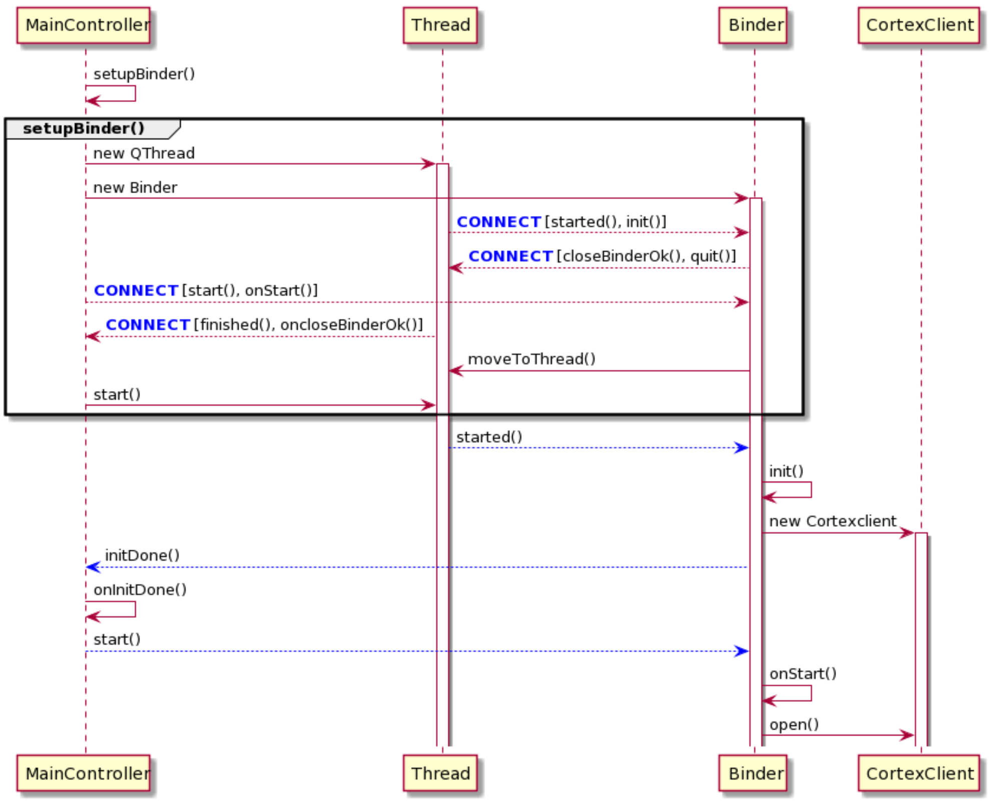


Figure 3.4: Creation of secondary thread

3.4.3 Front-end integration

The QML scripting language provides a fast way to prototype and create complex graphics but its capabilities can sometimes be limiting. Qt framework offers the possibility to extend the QML run-time engine with C++ functionalities, preferred in high computational cost situations. The sub-module businessLogic provides a special class called Bridge, able to interact with the QML engine. Thanks to the macro `PROPERTY()` the developer can expose QML interfaces in the form of signals and slot. Each property is composed by:

- **attribute name** to distinguish the property in the QML environment.
- **private variable** to store changes of the property.
- **setter method** to modify the attribute safely from both QML and C++ side.
- **getter method** to get the property value.
- **slot** to detect a change in the private variable.

The class is instantiated at the beginning of the program and after that inflated in the QML engine thanks to the method `setContextProperty()`. In this way, a Bridge global object is directly available in each QML script and each exposed property is accessible via the dot operator. Each time a property is modified the engine triggers the exposed slot to advise the back-end of the program.

On the other hand, the MainControler holds a reference to the Bridge object, and using the getter and setter methods can modify safely its properties since the engine can detect changes via the provided slot. The bridge properties were used to handle interactions with the graphic user interface, prompt warning, and error messages to the user.

```
Q_PROPERTY(
    QString promptMessage
    READ promptMessage
    WRITE setPromptMessage
    NOTIFY promptMessageChanged
)
```

Figure 3.5: Definition of the property `promptMessage` used to show warnings and errors in the bottom nav-bar.

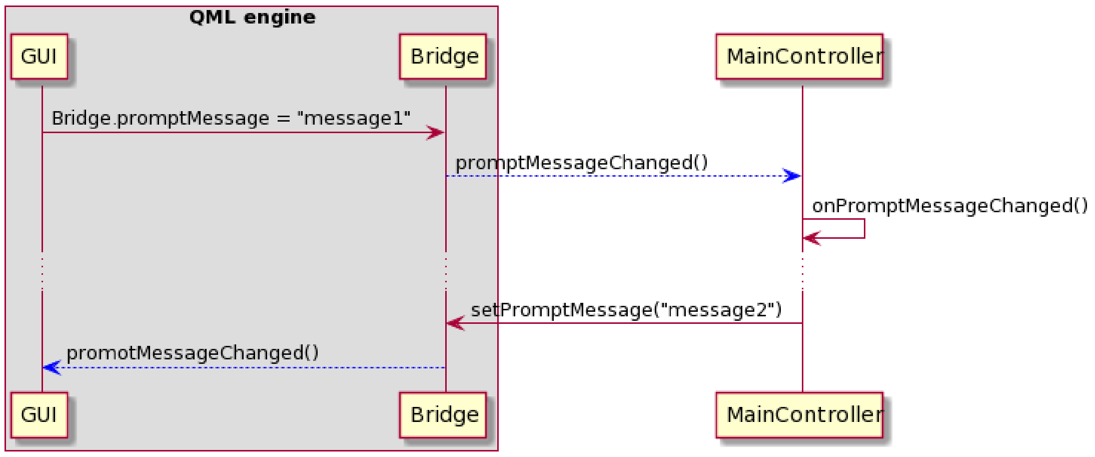


Figure 3.6: Interaction between the QML engine and C++ libraries. GUI element represents whatever qml object belonging to the `gui` sub-module.

3.4.4 Commands processing

The Xavier application collects several commands from the headset with a frequency of 8 Hz or 32 Hz, according to the selected set of trained features. These commands are then processed by the `CommandManager` class to be compliant with the Paravan system and filtered against several tuning indexes available in the user dashboard. The user is allowed to choose between two modalities: normal mode drive and assisted mode drive. In both, the user can manipulate the response intensity of brake and steering wheel, setting the following parameters:

- Steering wheel:
  - `Steering duration`, expressed in seconds
  - `Maximum angle` in module, expressed in degrees where the 0 represent the car going straight away
- Brake pedal:
  - `Brake duration`, expressed in seconds
  - `Max Brake intensity` expressed as a floating number in a range 0 1.

The command manager class, each time streaming begins, is fed with those parameters and produces two vectors, `vectorSteeringAngles`, and `vectorBrakeIntensity`, containing the cubic progressions of intensity that control the car behavior during the drive session following the given specifications.

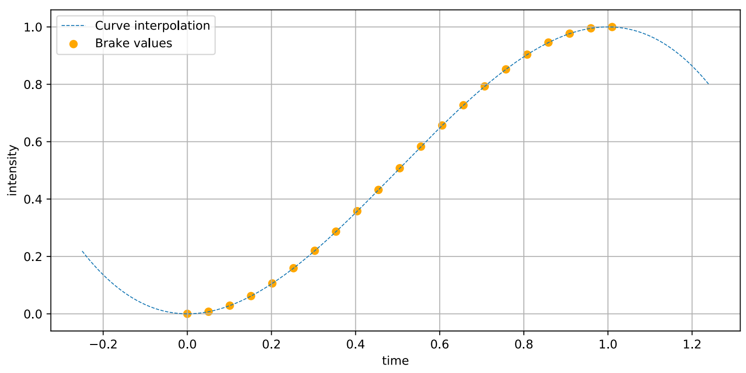
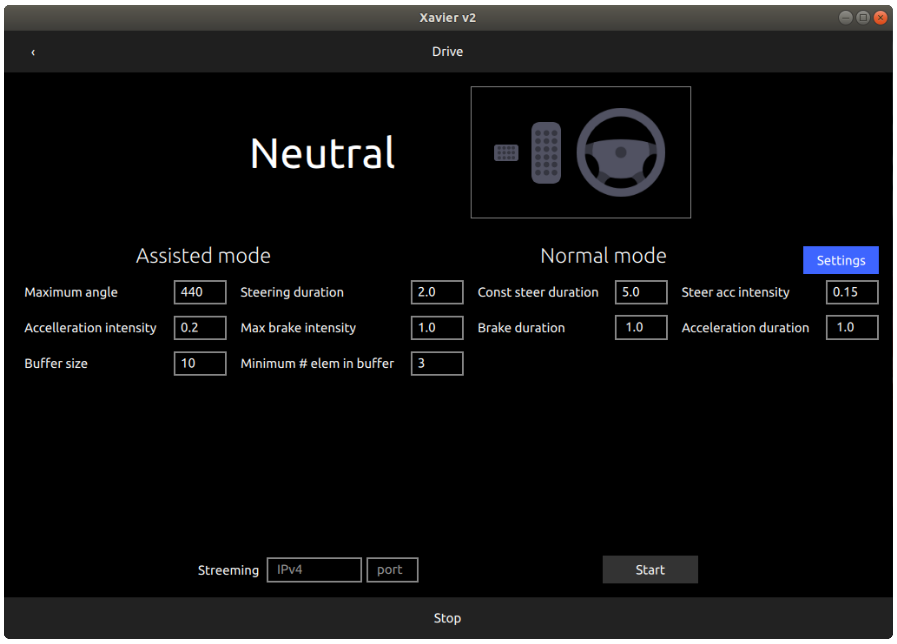


Figure 3.7: `vectorBrakeIntensity` with 1 second of duration time and 100% max intensity - each time the command Brake is detected the program follows this cubic function tho mitigate the brake intensity.

In normal drive mode, the detected commands are processed sequentially against the produced vectors to forward actions with different intensity. For instance, each time the brake command is held by the user the program iterate through the `vectorBrakeIntensity` to progressively increase the brake power. Once reached the end of the vector command manager holds the command with max intensity as long as the user holds the command.

Commands are stored in a circular buffer and filtered against the `getFilteredAction` function. Each new incoming command is pushed into the buffer, customize-able via the buffer size field, and at each iteration, the filter function returns the most relevant command in the buffer keeping consistent and smooth command transactions. The last two customize-able fields in normal drive mode are Steering acceleration intensity, which specifies the amount of throttle needed during a steering and acceleration intensity that locks the amount of throttle while going straight away. This mode is quite sensible to a lack of concentration and detection errors. On the other, hand is easy to configure and ready to go.



Assisted mode

Maximum angle	440	Steering duration	2.0
Accelleration intensity	0.2	Max brake intensity	1.0
Buffer size	10	Minimum # elem in buffer	3

Normal mode

Const steer duration	5.0	Steer acc intensity	0.15
Brake duration	1.0	Acceleration duration	1.0

Figure 3.8: Xavier application - assisted mode settings panel

In Assisted drive mode detected commands are still stored in a circular buffer but are filtered with a further parameter called `elementsThreshold` which represents the minimum number of occurrences in the buffer, for a given command to let it be detected. For instance, if we set this parameter with 3, each command has to be present at least 3 times to be sent to the car. The assisted mode drive supports all the features provided in the normal one plus three more parameters:

- Constant steering duration: the number of seconds to hold the steering position
- Acceleration duration: seconds to hold the acceleration in a straight way
- minimum number of elements in the buffer: that represents the command manager `elementsThreshold`

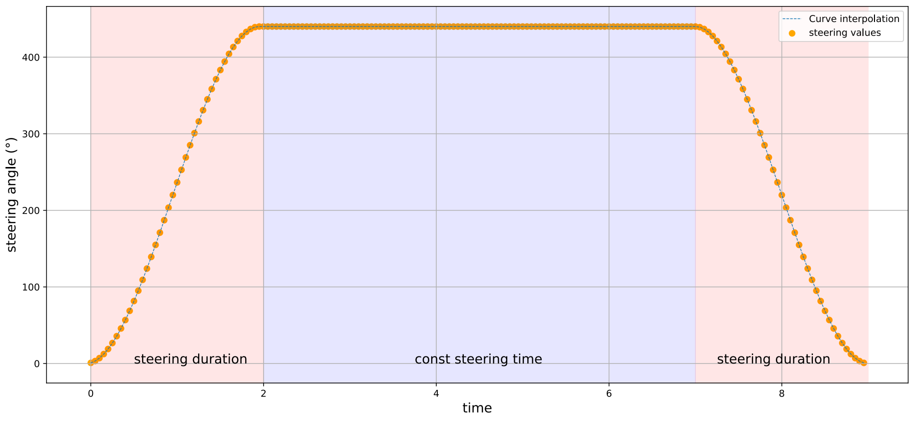


Figure 3.9: `vectorSteeringAngles` with 3 second of duration time and 5 seconds of const steering time

Chapter 4

Simulation

In the next part of this thesis project, a simulation environment has been provided to test the Xavier application in a safe way.

The simulators are composed of a Linux operating system, the robot operating system framework - ROS - and a collection of Python and C++ programs. Then simulator systems have been deployed using Docker technology.

In the following section, we will analyze the stack of technologies that compose the simulation and illustrate the judgment metrics with which to evaluate the quality of the Xavier BCI system.

4.1 Docker

Virtualization, the abstraction of physical hardware, could be very useful to wrap an application ensuring isolation. On the other hand, this technique suffers in performances since needs to include in a virtual machine a full copy of the required operating system, the application itself, and all the libraries and binaries.

Docker is a smart software solution for packaging and deploy code into standardized units called containers. Respect to virtualization, Docker offers a common infrastructure layer, letting containers to share OS dependencies and packing only the necessary for the application, assigning for each container unit an isolated lightweight process in user-space.

A system administrator can deploy oodles containers on the same machine, configure their ports as interfaces for clients as well as other containers, and ensure at the same time good performances, security among units, and scalability.

Nowadays docker and its containers represent a huge trend in application deployment and it is supported by the most relevant players in hosting business.

4.1.1 Architecture and deployment

Once properly installed, Docker is present in the form of a daemon program and it is in charge to administer containers. Furthermore, that daemon offers the possibility to interact with the Docker environment via a CLI application and a REST API.

Each running container is associated with a single process in the user space and can interact with its private file system only, stored in a so-called Docker image. Docker images include all needs by the application, binaries OS objects, and so on. Different containers can start from the same image, customize it, and produce an extended one for its purposes. Available official Docker images are collected on an online service called Docker hub.

The programmer can automatically deploy a Docker image with a script document called Dockerfile copying source codes, build them, install programs from package managers, and also expose ports.

4.1.2 Nvidia Docker

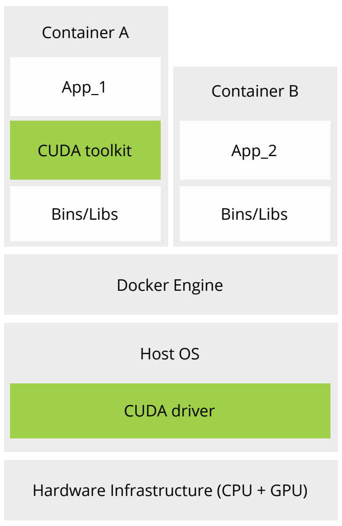


Figure 4.1: Docker architecture - Stack representation of the docker abstraction layer and two deployed applications. Boxes in green represent Nvidia drivers and tool-kits

Docker containers are remarkably efficient for service deployment and formal computation but can suffer in performances while used for graphics applications like 3D simulations. Intense graphics computation required special equipment called the graphics graphics process unit - GPU. A GPU will support the CPU addressing directly in hardware expensive parallel computation that involves matrices and arrays. The Nvidia company, a producer of GPU, offers a complete platform for parallel computing called CUDA - Compute Unified Device Architecture. Since Docker does not support GPU computation, a third-party toolkit has been developed by Nvidia to support its CUDA API to interface with their products. The toolkit its an open-source project available on the official GitHub page of the company and currently has support only Linux machines. For the scopes of that project version 2.0 has been used.

4.2 ROS

The robot operating system is an open-source project started by the Stanford Artificial Intelligence Laboratory in 2007, supported nowadays by the OSRF - Open Source Robot Foundation. ROS is a software platform that provides tools and libraries for hardware-independent robotic applications.

It offers a high degree of scalability and re-usability: ROS projects are structured as independent packages, a collection of run-able programs called nodes, capable to communicate within the system via a publisher-subscriber paradigm. It supports data communication among multiple operating systems allowing interaction within different hardware facilities. Besides it collects several tools for debugging and program development, all of them available with both command-line and graphical user interface and guarantees compatibility with the most relevant programming languages like C++, Python, Java, MATLAB and so on.

That framework has been shipped with two licensing plans, BSD and Apache, promoting collaboration among different projects and encouraging the growth of the platform itself.

4.2.1 Architecture

Once installed upon a Linux distribution, some steps are required to properly run ROS projects :

1. source ROS commands in `/opt/ros/<ros-version>/setup.sh`
2. run the command `roscore` in a separate terminal, which is a collection of programs that are mandatory for a ROS system.

The roscore is composed of three main components: the master node, a parameter server, and a logging node.

ROS master node is in charge to administrate the nodes, classifying them as publishers and subscribers, and holds their subscriptions to topics as well as services. It consists of an XMLRPC server that allows a node to locate one another. Once these nodes have detected each other they can communicate peer-to-peer without the interaction of the master. To do so that node exposes an XMLRPC-based API accessible via client libraries like `roscpp` or `rospy`.

The parameter node is used by nodes to keep track of configuration parameters at run-time. This component is directly connected to the master one and for this reason, its API is available in the same way through the XMLRPC protocol.

In the end, the rosout node is responsible to collect all the messages coming from the running systems and present them to the programmer within 5 verbosity levels:

- `DEBUG` debugging, a run-time inspection of variables
- `INFO` Meaningful information presented to the end-user
- `WARN` Situation that may lead to errors
- `ERROR` Fixable problems
- `FATAL` Unrecoverable problems that cause the end of execution

From now on the user can build packages stored in the `catkin_workspace` folder and run them via the CLI interface or launching a `roslaunch` file, a script document written using the XML language.

4.2.2 Rqt

Rqt is a wrapping framework around the already presented Qt that allows expanding nodes capabilities with flexible and customize-able graphic user interfaces. It is structured in three meta-packages: the `rqt` framework library itself, `rqt_common_plug`, and `rqt_robot_plugins`. Rqt custom plugins can be written in python or C++ and they provide GUI components via the legacy widget mode where interfaces are described via XML files.

The Rqt plugins also allow the user to interact directly with the ROS environment, manipulating perimeters, showing logs, plot data, control movements, send messages, and so on.

This library is very useful for fast prototyping and usually used to experiment with nodes at development time, before realizing the final structure of nodes. In this thesis project, the `rqt_graph` plugin has been very useful to debug and understand relationships between nodes and topics.

4.2.3 Rviz

Rviz is the ROS built-in 3D visualization tool, allowing us to easily visualize and interpret messages in a three-dimensional Cartesian plane. Thanks to the support to the Unified Robot Description Format - URDF - this tool is able to show a 3D representation of a machine that can be directly manipulated from the user according to the device description, and visualize the collected sensing information from robot's built-in devices like proximity sensors, LIDAR sensors, sonar sensors, cameras and so forth.

4.2.4 Gazebo

Gazebo is an open-source project supported by OSFR that offer a comprehensive 3D simulation environment. The application is shipped into two versions: A stand-alone version and a `gazebo_ros_pkgs` collection of packages which is a wrapper around the former. It supports several high-end physics engines like ODE, Bullet, and Simbody. The environment and objects in Gazebo are usually represented in the SDF open format, an XML-like textual format used to describe the objects in the space, the simulation environment, physical characteristics, and so forth.

4.3 Car_demo simulator

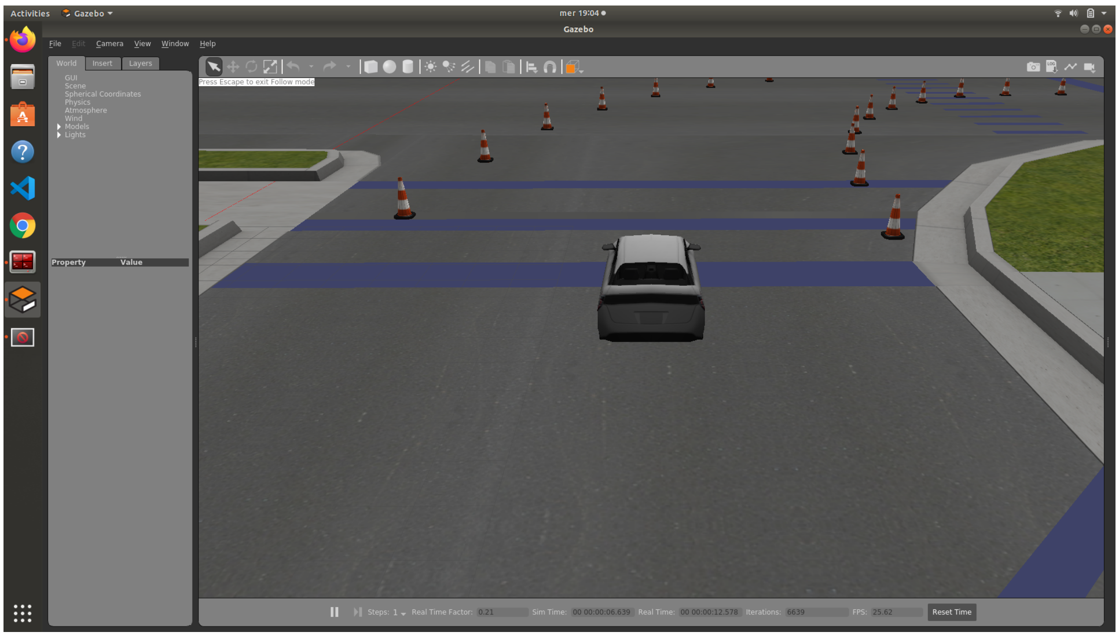


Figure 4.2: Car_demo - View of Gazebo with the car on the starting line

Car demo is an open-source project available on the official page of the OSRF foundation which allows us to set up a complete simulation environment for automotive applications. It is a mock-up ROS project structured in a way to easily deploy it in a container thanks to the Nvidia Docker technology.

The project offers a ready-to-run simulation of a Toyota Pius car, equipped with cameras, Lidars, and proximity sensors. It is composed of three ROS packages, a dockerfile script, and two bash scripts for building and deployment procedures.

`Prius_description` contains all the necessary scripts to describe the car structure, behavior, and appearance in the simulator. The car is rendered based on several mesh files in png format and the overall car structure and equipment are described in the `prius.urdf` script.

`Prius_msgs` defines the main interface with which the Toyota car accepts messages. The car has 3 gears expressed as eight-bit integer numbers - neutral, forward, and reverse - while throttle, brake, and steering are expressed as sixty-four-bit floating numbers.

The last and main package is called `car_demo` and contains:

- A Ros node used to interpret messages coming from a controller
- The Rviz configuration file
- The description of the world rendered by Gazebo
- A launch file used to run all the nodes required for the project

.

For our purposes, that mock-up project has been consistently modified to produce an efficient and light-weight simulator used to test maneuvers with the car controller via the Xavier application and keep notes about the times.

4.3.1 Project customization and final architecture

The first modification applied to the mock-up was to remove all the unnecessary. The car description file has been reduced to the only part that belongs to the chassis and mechanical components, all the sensors have been removed to decrease the computational effort made by the hosting machine and the Rviz panel has been disabled from the ROS launch file as well as the joystick translator. Besides, a following point-of-view where added to let the user control the car having always the vehicle in the center of the screen.

In the end, the project has been extended with further ROS packages:

- `cmd_prius_w_msgpack` that emulates the Paravan system, It is composed of a simple UDP client and a C++ ROS publisher that converts messages into ones suitable with the standard accepted by the simulator.
- a `stopwatch` script written in Python able to interact with the ROS environment.
- `tf_watchdog`, a collection of python scripts in charge of rendering a path in the simulator and check the car position in the map

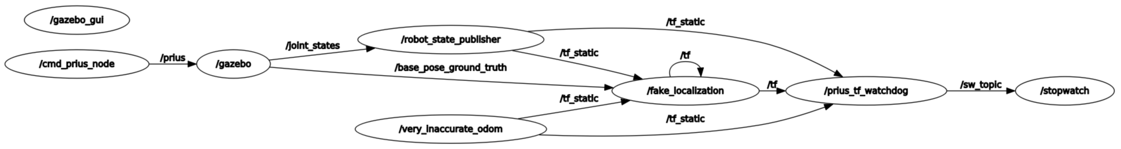
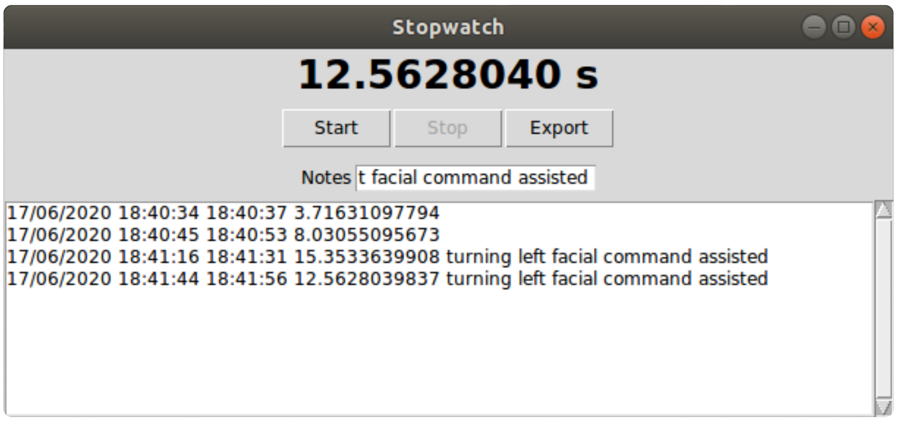


Figure 4.3: Car_demo - Final structure of nodes, each node communicate within the system thanks to the topics, the arrows.

4.3.2 Stopwatch ROS package



```
17/06/2020 18:40:34 18:40:37 3.71631097794
17/06/2020 18:40:45 18:40:53 8.03055095673
17/06/2020 18:41:16 18:41:31 15.3533639908 turning left facial command assisted
17/06/2020 18:41:44 18:41:56 12.5628039837 turning left facial command assisted
```

Figure 4.4: The Stopwatch application

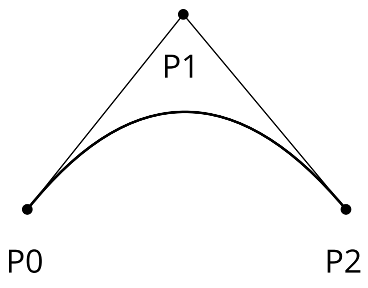
The stopwatch is an application written in Python that lets the user record time intervals using the system clock, to take some notes for each recording and also allow to export a whole recording session into a CSV file. The graphic user interface has been developed via the `Tkiniter` library and present two buttons for start and stop the recording, an export button used to export info into CSV file, an input field box that let the user take notes about the recordings and a large output field for logging purposes.

Furthermore, the stopwatch application offers an extended version of the regular application which allows this node to define a ros-topic and listen for messages. In this way, the start and stop commands can be triggered by others nodes in the system simply publishing a message to that topic.

4.3.3 Watchdog ROS package

That ROS module is composed of two python scripts, the cone spawn-er and the watchdog itself.

The cone spawn-er is in charge to render a predefined path in Gazebo rendering some road cones in the virtual environment. This module automatically computes a quadratic Bézier curve, a particular parametric curve used in computer graphics.



$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \ , \ 0 \leq t \leq 1$$

$$= (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2 \ , \ 0 \leq t \leq 1$$

Figure 4.5: Given the points P0 P1 and P2, a quadratic Bézier curve is described by the function B(t)

Each path is composed by two parallel curves, saved in a dictionary specifying the three points and the cones are spawned accordingly using the code snippet depicted below.

```
def compute_trajectory(p0,p1,p2):
    path_x = []
    path_y = []
    dt = np.arange(0,1,0.1)
    for t in dt:
        # b(t) = (1-t)^2*p0 + 2t(1-t)*p1 + t^2p2, t in [0,1]
        # a = (1-t)^2
        # b = 2t(1-t)
        # c = t^2
        a = (1-t)*(1-t)
        b = 2*t*(1-t)
        c = t*t
        x = (a*p0.x) + (b*p1.x) + (c*p2.x)
        y = (a*p0.y) + (b*p1.y) + (c*p2.y)
        path_x.append(x)
        path_y.append(y)
    return path_x,path_y
```

The watchdog node itself is in charge to check the car position on the map in gazebo, trigger the stop command of the stopwatch when the finish line has been reached and re-positioning the car on the starting line in order to run another drive session. This goal has been achieved thanks to the transformation `/tf` topic of ROS. The ROS tf package is used to keep track of multiple coordinate frames over time and allow to transform points and vectors.

An instance of the `TransformListener` class has been instantiated in the script to subscribe to ROS transform messages via the `lookupTransform()` method. This function will return lists: the linear transformation of the child frame relative to the parent and the quaternion required to rotate from the parent orientation to the child orientation.

4.3.4 Building and running procedure

The whole project has been built via the the Dockerfile. Each command in that script file represent a layer. The Docker system keeps track of each layer computing its hash in a way that layers already computed can be cached, speeding up the entire building procedure.

```
# official  ROS image hosted on dockerhub by the OSRF foundation
FROM osrf/ros:kinetic-desktop

RUN apt-get update \
 && apt-get install -y \
    wget \
    lsb-release \
    sudo \
    mesa-utils \
 && apt-get clean



# Get gazebo binaries
RUN echo "deb http://packages.osrfoundation.org/gazebo/ubuntu `lsb_release -cs` main" > /
     etc/apt/sources.list.d/gazebo-stable.list \
 && wget http://packages.osrfoundation.org/gazebo.key -O - | apt-key add - \
 && apt-get update \
 && apt-get install -y \
    gazebo9 \
    ros-kinetic-gazebo9-ros-pkgs \
    ros-kinetic-fake-localization \
 && apt-get clean



```

```
26 # Get msgpack http s://msgpack.org/
27 # It is used to parse UDP packets coming from the Xavier application
28 RUN git clone https://github.com/msgpack/msgpack-c.git \
29   && cd msgpack-c \
30   && cmake . \
31   && make \
32   && sudo make install \
33   && cd ..
34 
35 # setup the ROS workspace environment under the /tmp folder
36 RUN mkdir -p /tmp/workspace/src
37 COPY prius_description /tmp/workspace/src/prius_description
38 COPY prius_msgs /tmp/workspace/src/prius_msgs
39 COPY car_demo /tmp/workspace/src/car_demo
40 COPY tf_watchdog /tmp/workspace/src/tf_watchdog
41 COPY stopwatch /tmp/workspace/src/stopwatch
42 COPY cmd_prius_w_msgpack /tmp/workspace/src/cmd_prius_w_msgpack
43 RUN /bin/bash -c 'cd /tmp/workspace \
44  && source /opt/ros/kinetic/setup.bash \
45  && catkin_make'
46 
47 
48 CMD ["/bin/bash", "-c", "source /opt/ros/kinetic/setup.bash && source /tmp/workspace/
       devel/setup.bash && roslaunch car_demo demo.launch"]
```

After the building porcedure a `car_demo` image has been produced and can be launched using the `./run_demo.bash`. That scripts is able to run the container via the `rocker` program, provided by OSRF, that help to run graphic programs within a container.

```
rocker --nvidia --x11 \
 --oyr-run-arg "--name=car_emulator -p 6071:6071/udp" \
 -- osrf/car_demo
```

The nvidia parameter specifies that the container has to be triggered with CUDA enabled, the name parameter defines the name assigned to the generated container while the p parameter designates the ports mapping between host machine and container. The last parameter is required to properly forward UDP messages coming from Xavier to the container, reaching the `cmd_prius_w_msgpack` node. After launching the container it can be reached for further analysis running a terminal session via the command: `docker exec -it car_emulator bash` or terminate the container via `docker kill car_emulator`.

The last command specified on the Dockerfile at row forty-eight, once the container has be ran, will source the ROS commands, the built packages in the workspace directory and finally launch the `demo.launch` script

```
1  <?xml version=”1.0”?>
2  <launch>
3      <arg name=”model” default=”$(find prius_description)/urdf/prius.urdf”/>
4      <param name=”robot_description” textfile=”$(arg model)”/>
5
6      <include file=”$(find gazebo_ros)/launch/empty_world.launch”>
7      <arg name=”verbose” value=”false”/>
8      <arg name=”world_name” value=”$(find car_demo)/worlds/mcity.world”/>
9      </include>
10
11     <node pkg=”robot_state_publisher” type=”robot_state_publisher” name=”
       robot_state_publisher”/>
12     <node pkg=”fake_localization” type=”fake_localization” name=”fake_localization”/>
13     <node pkg=”tf2_ros” type=”static_transform_publisher” name=”very_inaccurate_odom”
       args=”0 0 0 0 0 0 odom base_link”/>
14
15     <node name=”spawn_urdf” pkg=”gazebo_ros” type=”spawn_model” args=”−param
       robot_description −urdf −x 3 −y −12 −z 0.5 −model prius”/>
16
17     <node pkg=”tf_watchdog” type=”watchdog.py” name=”prius_tf_watchdog”>
18         <param name=”/use_sim_time” value=”true”/>
19     </node>
20     <node pkg=”cmd_prius” type=”cmd_prius_node” name=”cmd_prius_node”/>
21     <node pkg=”tf_watchdog” type=”cone_spawner.py” name=”cone_spawner”/>
22     <node pkg=”stopwatch” type=”stopwatch.py” name=”stopwatch”/>
23
24 </launch>
```

The first half of the script, from line 3 up to line 16, is the legacy one in charge of launching the Gazebo application, setup the car model, spawn it in the simulation environment and enable the model tracing withing ROS.

The second half instantiate all the custom nodes that explained in the previous sections: the stopwatch, the watchdog, the cone spawn-er, and the Paravan system emulator.

Chapter 5

Testing and conclusions

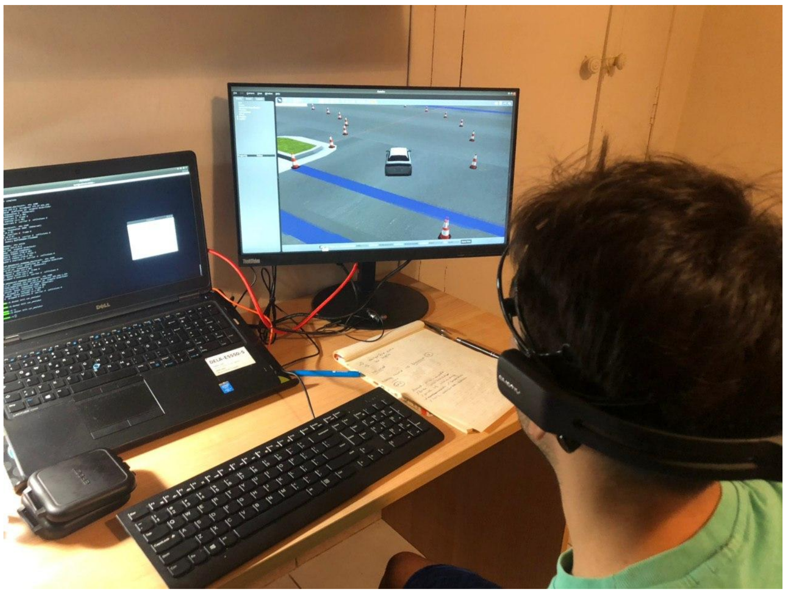


Figure 5.1: The Stopwatch application

In this final chapter we will present all the results obtained by the whole system, the training procedure, the drive simulations, and some results in terms of accuracy, exploiting the different driving combinations: facial expressions or mental commands, normal drive mode, or assisted mode. In the end, we will trace some conclusions about the goals achieved by the project, some proposals for the next iteration of the Xavier application, and the expectations for the further developments of the entire project.

5.1 Testing with car_demo

After the release of the Xavier application and the simulator, several free drive sessions were conducted in order to judge the overall response of the car to the headset inputs and let the user be enough confident with the system. In the end, more formal tests have been conducted.

All the tests were attended by a single subject since the training procedure, the environment setup, and taking confidence with the headset is quite time-consuming.

5.1.1 Training

Before the actual ride, the BCI system requires a training session that has to last $\approx$ 20 minutes per mental commands and a few minutes for facial expressions to be effective, as suggested by the official Emotiv guide. The system does not restrict the kind of signals that can be used in any way. Each user is free to use whatever type of tough, with the only requirement that it has to be reproducible. Usually the most frequent inputs are tough concerning the movement of objects or body parts like arms or feet.

The training has been performed via the official Emotiv application.

5.1.2 Tests

Tests have been divided into two so-called drive sessions; One with Facial expressions has been last $\approx$ 45 minutes while the one with Metal commands $\approx$ 2 hours.

For each drive session, has been asked to the user to run as much as ride as possible, in both left-curve path and right-path, using assisted and normal mode. For each combination 5 ride has been selected and assessed. The results are reported in Table 5.1, Table 5.2, Table 5.3, and Table 5.4. In the end, means and standard deviation have been computed and reported in Table 5.5

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Figure 5.2: Arithmetic mean of n elements $x_1 \cdots x_n$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - m)}{n}}$$

Figure 5.3: Standard deviation of n elements $x_1 \cdots x_n$

Table 5.1: Facial expression normal mode

	Path	Time (s)
1	Left curve	17.032
2	Left curve	19.041
3	Left curve	23.347
4	Left curve	26.676
5	Left curve	31.244
1	Right curve	16.637
2	Right curve	20.168
3	Right curve	21.041
4	Right curve	23.111
5	Right curve	24.593

Table 5.2: Facial expression assisted mode

	Path	Time (s)
1	Left path	28.666
2	Left path	31.462
3	Left path	37.112
4	Left path	37.501
5	Left path	40.664
1	Right path	28.932
2	Right path	31.341
3	Right path	32.52
4	Right path	37.989
5	Right path	39.396

Table 5.3: Mental commands normal mode

	Path	Time (s)
1	Left curve	164.824
2	Left curve	170.848
3	Left curve	213.579
4	Left curve	234.832
5	Left curve	269.069
1	Right curve	152.619
2	Right curve	153.743
3	Right curve	203.291
4	Right curve	213.057
5	Right curve	275.448

Table 5.4: Mental commands assisted mode

	Path	Time (s)
1	Left path	196.263
2	Left path	224.825
3	Left path	234.808
4	Left path	256.426
5	Left path	280.094
1	Right path	172.905
2	Right path	176.985
3	Right path	190.464
4	Right path	219.246
5	Right path	241.612

Table 5.5: Means and standard deviations

BCI mode	Drive mode	Mean	Standard deviation
Facial exp	Normal	22.289	4.275
Facial exp	Assisted	34.559	4.220
Mental com	Normal	205.131	42.720
Mental com	Assisted	219.363	33.389

5.2 Tests assessment

The results of the tests shown a significant difference in time performances between driving with Facial expressions and Mental commands; The paths have been covered in an average time of 28,42390959 seconds with facial commands against the 212,246977 seconds with mental ones.

Indeed, the implementation of the assisted mode algorithm has been judged looking at the average time covered against rides performed in normal mode, with both mental commands and facial expressions. The data depicted in Figure 5.4 and Figure 5.5 shown better results in favor of normal mode in both drive sessions. Further investigations have been conducted to statistically prove the meaningfulness of that assessment.

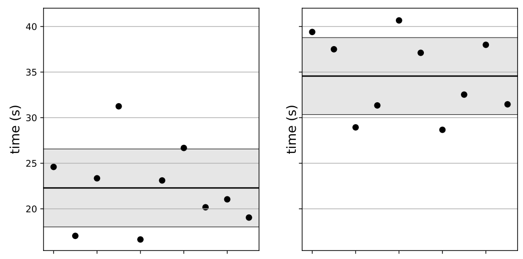


Figure 5.4: Facial expression test results - distribution, mean and standard deviation of tests conducted in normal mode (on the left) and assisted mode (on the right)

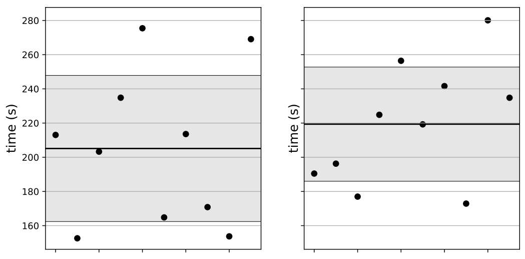


Figure 5.5: Mental commands test results - distribution, mean and standard deviation of tests conducted in normal mode (on the left) and assisted mode (on the right)

5.2.1 Student's t-test

The Student's t-test is a statistical tool used to tell if there is a significant difference between the means of two groups. We used that tool to assess that the higher average (less performance) obtained in the assisted mode is due or not to randomness.

The judgment process starts defining a *null hypothesis* H_0, which states that there is no statistical difference between the means of two populations, and an H_1 hypothesis that states the opposite.

$$\begin{cases} H_0 : \mu_1 - \mu_2 = 0 \\ H_1 : \mu_1 - \mu_2 \neq 0 \end{cases}$$

The test will produce a t-statistic t_s value which is the fraction of the difference between sample means and the pooled standard deviation s_{12}

$$t_s = \frac{(\overline{X_1} - \overline{X_2}) - (\mu_1 - \mu_2)}{s_{12}\sqrt{\frac{n_1+n_2}{n_1 n_2}}} \xrightarrow{H_0} t_s = \frac{\overline{X_1} - \overline{X_2}}{s_{12}\sqrt{\frac{n_1+n_2}{n_1 n_2}}}$$

$$s_{12} = \sqrt{\frac{(n_1 - 1){s_1}^2 + (n_2 - 1){s_2}^2}{n_1 + n_2 - 2}}$$

T-statistic can be interpreted by comparing it to critical values from the t-distribution. The critical value can be calculated using the degrees of freedom and a significance level with the percent point function (PPF).

In the end the *null hypothesis* can be accepted if and only if

$$|t_s| \leq t_c$$

These test has been conducted using python and the open-source libraries scipy and numpy, with a significance level of 0.05 - 95% of confidence.

```python
def ttest_eval(a,b,alpha=0.05):
    t_statistic , p_value = stats.ttest_ind(a,b)
    # degrees of freedom
    df = len(a) + len(b) - 2
    t_critical = stats.t.ppf(1.0 - (alpha/2), df)
    print('ts: %s' % t_statistic)
    print('tc: %s '% t_critical)
    if abs( t_statistic ) <= t_critical):
        print('[abs(ts) <= tc] Accept null hypothesis that the means are equal.')
    else:
        print('[abs(ts) > tc] Reject the null hypothesis that the means are equal.')
```

```
>>> import data as d
>>> d.ttest_eval(d.facial_normal, d.facial_assisted)
ts: -6.12816409020712
tc: 2.10092204024
[abs(ts) > tc] Reject the null hypothesis that the means are equal.
>>> d.ttest_eval(d.mental_normal, d.mental_assisted)
ts: -0.7874355997696234
tc: 2.10092204024
[abs(ts) <= tc] Accept null hypothesis that the means are equal.
```

As we can see the *null hypothesis* was rejected for the two facial groups, meaning that they belong to different means i.e we can confirm that the assisted mode affected negatively the performances of the driver.

On the other hand, the test applied to the mental groups confirms the *null hypothesis* meaning that we cannot judge as separate results the two means since they belong to the same distribution with the same global mean μ.

In the end, the assisted mode seems to decrease the system performances using facial expressions, slowing down the overall reaction while we do not observe the same issue driving with mental commands, but at the same time, that mode does not increase the ease-of-use and reliability of the system.

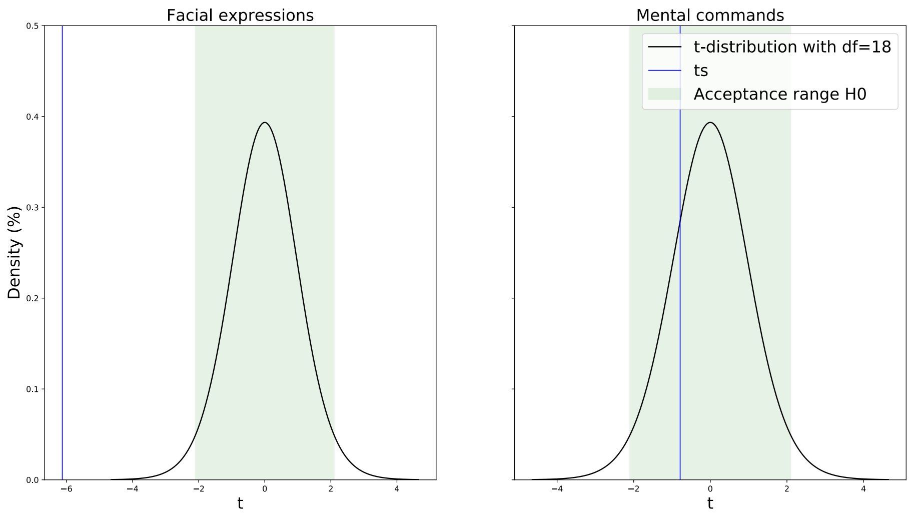


Figure 5.6: t-test graphic representation - facial expressions on left, mental commands on the right. The green zone represents the acceptance region of the H_0 hypothesis between [-tc,tc]

5.3 Final considerations

The third iteration of the Xavier project is more mature and robust, with the possibility to extend and scale the project in the future, but it is not exempt from problems and some issues have been found an presented in the following sections.

In conclusion, the Xavier application could be considered as a starting point for the development of more sophisticated algorithms, that have to be tested with a larger number of testers.

5.3.1 Operating system

Cortex API is (currently) available for personal computers only with Windows and MacOSx. Consequently, the system needs a secondary x86 architecture machine directly installed into the car to run the Xavier program and interact with the Headset. Cannot run the application on a Linux OS system is quite limiting since the majority of the infotainment car's systems relies on it. In general, the impossibility to run that code on a RISC system on chip discourage the usage of the Emotiv system in resource-aware systems.

On the other end, the company is releasing many updates for the system and recently it has also launched a mobile application to experiment with the helmet. This should suggest the company's interest in mobile/RISC development and a further version of the Cortex API compatible with these systems.

5.3.2 Headset maintenance

One of the crucial points of the system was the headset itself. The Emotiv Epoc+ device has a lot of pros respect other products available in the market but it is not free from defects.

The principal point of failure of the Epoc+ is the electrode. These compotes need to bee attached and removed for each session with the system and treated with the saline solutions. The following problems have appeared emerged with them:

- The locking mechanism of a single electrode is composed of a thin piece of plastic used to anchor the component on the root of one of the mounting points. This piece is way too unreliable for common use for a driver since it is brake prone. Once broken the lock mechanism, the electrode is not able to keep a good connection with the headset, affecting the results of the monitoring or even worst the impossibility to set the electrode in place. The Epoc+ headset must have all the electrodes in place to run a session since missing electrodes cause noise and they will stain the overall recording.

- Each electrode presents a small coated metal plate used to transmit bio-potentials received from the soaked felt. After an intense use of the headset for several months, the plates started to being oxidized. This issue is reported by the user manual of the headset, saying it should not affect the performances.
- Electrodes are soaked before each ride with a common saline solution available on the market. The system will ensure a good capture of the signals if and only if all the electrodes are soaked enough. In the longest test sessions, some tests were judged unreliable due to the low precision of the Epoc+ device caused by drying electrodes. This phenomena also affect the ease-of-use of the system since once removed all the electrodes and re-positioned in place, the user should train again commands for better results.

Further issues came along with the charge of the headset; during several tests, we experienced a lack of precision when the Epoc+ is not fully charged. It is also unusable if the charge is lower than 25%.

Indeed long driving sessions (2 hours and more) with the headset may cause headaches and itch on the head of the subject.

Nowadays commercial BCI solutions are not so expansive and the market offers a lot of different solutions: An headset with fewer electrodes like the Epoc one, from the same company, could be much easier to use, already compatible with our system, more accessible, better power consummations but less accurate ad with same issues about electrodes. A completely distinct solution could be a headset with fixed electrodes which does not require a saline solution, like the Muse two headset. Unfortunately, this device currently does not offer any public API for developers and presents only a few channels in the 10-20 positional system, being poorly accurate for our purposes.

Figure 5.7: Muse 2 headset

5.3.3 Driving algorithm

During the tests, we notice good results and performances with facial expression, given by both user satisfaction and times per lap. We also demonstrate the inefficiency of the assisted mode using facial expressions statistically but we also hi-light the huge time gap between facial expression and mental commands driving sessions.

We suggest for future developments to exploit different algorithm strategies and new headset features, combined to enhance the driving experience, and to find further testing tetchiness that could speed up the testing process in Gazebo.

Possible solutions could be to pair the headset with other sensing devices like a camera and a Lidar to assist the driver with some ADAS-like systems. The car_demo emulator already provides all the necessary to explore in that direction.

Indeed we propose to study the possibility to use raw data from the headset and train by our self a neural network since the Cortex API, as closed source code, does not give us a real comprehension of what is behind the training and live mode.

5.3.4 Xavier application

At the end of the project, the Xavier application itself has achieved a higher level of maturity respect the predecessors, but during the intense test sessions we registered some issues:

- Despite the huge step over in the HMI, some use cases like the training has to be re-designed since they don't give the necessary feedback to the user. We experienced that a training session can be more engaging if the user could have visual feedback from the program, like a moving cursor or object. The same strategy has been applied by the latest official Emotiv applications.
- The version 1.0 of the Xavier application had serious problems with dangling pointers due to a bad implementation of the framework. The new version fixes this problem but it showed some issues concerning code design, also in terms of memory allocation. We suggest for for further implementations to re-design the two thread interaction and to reconstruct the linking between back-end components

Bibliography

[1] J. Richard Caton. «The electric currents of the brain». In: *British Medical Journal* 2 (1875).

[2] Mario Tudor. «Hans Berger- the history of electroencephalography». In: *Acta Med Croatical* 59,4 (2005), pp. 307–313.

[3] J. Kamiya. «Conscious Control of Brain Waves». In: *Psychology Today* 1 (1968), pp. 56–60.

[4] Henry Gray. *Gray's Anatomy of the Human Body.* London, GB: John William Parker, 1858 (cit. on p. 3).

[5] Martin Strmiska Zuzana Koudelková. «Introduction to the identification of brain waves based on their frequency». In: *MATEC Web of Conferences* (2018).

[6] Emotiv co. *The Introductory Guide to BCI.* URL: `https://www.emotiv.com/bci-guide/`.

[7] Cassidy AndrewJoseph. *Mastering Mental Commands.* URL: `https://www.emotiv.com/knowledge-base/training-mental-commands/`.

[8] Emotiv co. *Cortex v2 API.* URL: `https://emotiv.gitbook.io/cortex-api/`.

[9] Qt co. *Qt and QML 5.15 API.* URL: `https://doc.qt.io/qt-5/reference-overview.html`.

[10] Docker co. *Docker documentation.* URL: `https://docs.docker.com/`.

[11] Nvidia co. *Nvidia-docker poroject.* URL: `https://github.com/NVIDIA/nvidia-docker`.

[12] Open robotics. *Ros wiki.* URL: `http://wiki.ros.org/Documentation`.

[13] Open robotics. *Car_demo.* URL: `https://github.com/osrf/car_demo`.

[14] Gianmarco Altoè. *Notes from course: Corso di Psicometria Progredito.* URL: `https://people.unica.it/gianmarcoaltoe/files/2012/04/lezione4.1_test_t.pdf`.