

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA MECÁNICA



**“EXTRACTION AND ANALYSIS OF
CAR DRIVING DATA VIA OBD-II”**

TRABAJO FIN DE GRADO
Septiembre - 2018

BACHELOR THESIS
September -2018

AUTOR/AUTHOR: Fernando Bocanegra Gallardo
DIRECTORES/DIRECTORS: Prof. César Fernández Peris
Prof. Roberto Setola

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA MECÁNICA



**“EXTRACTION AND ANALYSIS OF
CAR DRIVING DATA VIA OBD-II”**

TRABAJO FIN DE GRADO
Septiembre - 2018

BACHELOR THESIS
September -2018

AUTOR/AUTHOR: Fernando Bocanegra Gallardo
DIRECTORES/DIRECTORS: Prof. César Fernández Peris
Prof. Roberto Setola

A mi familia y compañeros del laboratorio por haberme animado y ayudado con este proyecto, en especial a mis padres, a Luca y al Prof. César.

Alla mia famiglia e colleghi di laboratorio per avermi supportato e aiutato con questo progetto, specialmente ai miei genitori, a Luca e al Prof. César.

To my family and lab partners who encouraged and helped me to develop this project. Especially my parents, Luca and Prof. César.

CONTENT

1. MOTIVATION AND OBJECTIVES

2. STATE OF THE ART

2.1 BUS SYSTEMS

2.2 PROTOCOLS

2.2.1 CAN BUS

2.2.2 MOST

2.2.3 FlexRay Bus

2.2.4 SAE J1850

2.2.5 OBD-III Standard

2.2.6 Keyword

2.2.7 Local Interconnect Network (LIN)

3. AVAILABLE TOOLS

3.1 SOFTWARE

3.1.1 Delphi DS150

3.1.2 DashCommand

3.1.3 EOBD Facile

3.1.4 OBD Auto Doctor

3.1.5 Movi Pro

3.1.6 AutoEnginuity ScanTool

3.1.7 Vehicle Spy

3.1.8 PyOBD

3.1.9 Total OBD & ECU Auto Diagnostics (TOAD)

3.1.10 PCMSCAN

3.1.11 CANiBUS Server

3.1.12 Kayak

3.1.13 Octane

3.2 HARDWARE

- 3.2.1 ELM
- 3.2.2 Delphi
- 3.2.3 Arduino
- 3.2.4 VSCOM
- 3.2.5 Higher-End Hardware

4. EXPERIMENTS CARRIED OUT

- 4.1 THE AMBULANCE**
- 4.2 THE TEACHER'S PERSONAL CAR**
- 4.3 PROOF WITH THE RENTED CAR**
- 4.4 PROOF WITH LUCA'S CAR**

5. ANALYSIS

5.1 DATA WE CAN OBTAIN FROM THE VEHICLE

- 5.1.1 Frame Number
- 5.1.2 Frame Time
- 5.1.3 Forward and Lateral Acceleration
- 5.1.4 Vehicle Roll and Pitch
- 5.1.5 GPS Latitude, Longitude and Altitude
- 5.1.6 Engine Coolant Temperature
- 5.1.7 Intake Manifold Absolute Pressure
- 5.1.8 Engine Revolutions
- 5.1.9 Vehicle Speed Sensor
- 5.1.10 Intake Air Temperature
- 5.1.11 Air Flow Rate from Mass Air Flow Sensor
- 5.1.12 Fuel Economy Calculations
- 5.1.13 Trip Computer Calculations
- 5.1.14 CO₂ Content
- 5.1.15 Fuel Density
- 5.1.16 Stoichiometric air/fuel ratio

- 5.1.17 Mass Air Flow
- 5.1.18 Manifold Absolute Pressure
- 5.1.19 Fuel Flow
- 5.1.20 Average Fuel Flow
- 5.1.21 Boost Pressure Estimation
- 5.1.22 Trip Time Clock
- 5.1.23 Distance Travelled
- 5.1.24 Current Acceleration
- 5.1.25 Fuel Consumed

5.2 DATA ANALYSIS WITH MATLAB

5.3 ALGORITHMS DEVELOPED

5.4 RESULTS

6. CONCLUSIONS AND FUTURE WORK

7. BIBLIOGRAPHY

1. MOTIVATION & OBJECTIVES

When I applied for the Erasmus Program, I was supposed to develop my bachelor thesis abroad. Firstly, I had no idea of what to do and why. However, the idea of this project started when Prof. Setola offered me to work in the automatics lab of the university and aimed me to develop a thesis about the automatic control units that are inside vehicles. Although my training and branch of knowledge is mechanics, the fact of that “an engineer must learn about everything” encouraged me to carry on and accept this task. Since then, I have not stopped learning and nowadays I can no longer see a simple car and avoid thinking in the automatics that are inside of it.

The main objective of this thesis is the study of all data that can be extracted from a vehicle’s electronic control unit. In order to extract the data, several hardware will be used as well as some software. In particular, Dashcommand app will be used to extract all the “raw data” and Matlab will be used for the study. Moreover, the process will count with the help of two tutors: Prof. Roberto Setola from Università Campus Bio-Medico di Roma (mainly for the extraction of data) and Prof. César Fernández from Universidad Miguel Hernández (mainly for the study & development of this report).

Once data are extracted and can be seen, we will try to create some algorithms with the most interesting variables in order to get some evaluation of the driving. These algorithms will be developed using Matlab and my programming skills. Then, when the algorithms are finished, they will have to be able to evaluate the trip and answer the user with a driving mark according to how safety/efficient it was. Curiously, this idea is almost the same of other projects called *Drivies* or *Carmetry*, two spin-offs launched by *Telefonica’s I+D Department* or the UMH respectively. This last one is an app that allows to control fuel consumption, geolocate an accident or improve user’s driving in general. Here, thanks to some algorithms the app will tell users what to do and improve to reduce emissions, save petrol or travel safer.

2. STATE OF THE ART

In the last decades automotive industry is one of the sectors which have changed the most. Innovation has not stopped so every single model that has been developed has been made with new features that his previous versions did not have. For example, nowadays there are no longer models with manual windows, something that used to be in every single car just 20 years ago. But not only these little comfort features have been improved, also other computers were implemented for the safety (like the ABS break system or belts), the fuel consumption or other comfort systems such as the air conditioned. And all this has been possible with the help of the different electronic control units that are in the car. Today a new simple car has up to 30 electronic control units (usually called ECUs) which monitor the different variables.

An ECU takes care of controlling one or several electronic systems in a vehicle with the help of a microcontroller. This has a software and is able to make calculations and send an order according to the data it receives. Usually, each ECU works independently but there are some complex tasks that require the involvement of different ECUs. In fact, **the communication between the different subsystems in the vehicle is essential for its correct functioning**. In addition, most of the modern ECUs have a memory to save all the important data. This means when a mechanic wants to know what is happening in the car or what is wrong, he first connects to the car and extracts the information saved. That is a great different way to work in comparison with 20 years ago. Now everything is computerized and can be controlled thanks to the electronics of the car.

2.1 BUS SYSTEMS

As we have seen before, for the correct functioning and operation of a vehicle it is necessary to have a communication system that connects the different ECUs. At the beginning, in electronic systems signals were sent from one chip to another using **wires** (for the moment let us forget about wireless things). The simplest way of doing so is to use one wire per bit of information we want to transmit. One bit of information is simply an answer to a yes/no question like "*Are the headlights on?*" If the headlights are on, there is a voltage on that wire, usually 5 volts. If they are off, there are 0 volts on the wire.

Now that is fine for one bit of information, however more data require more wires. But unfortunately, more wires means more complexity. We can say **a modern car is just a computer with tires on it**, so there are a lot of wires (in fact several km) in it. More wires result in more weight and more costs, and car manufacturers always try to avoid this. Therefore, a way to reduce the amount of wires is needed.

The usual way of doing this is to use a **bus system**. In order to understand what a bus system is, we may think of a bus as a way to **transmit more information using fewer wires**. There are other benefits of bus systems, but for the moment we will focus on this aspect. For example, if we want to switch four lamps on or off:

Without bus system:

- Lamp 1: +5 V on Wire1 = Lamp is on; 0 V on Wire1 = Lamp is off
- Lamp 2: +5 V on Wire2 = Lamp is on; 0 V on Wire2 = Lamp is off
- Lamp 3: +5 V on Wire3 = Lamp is on; 0 V on Wire3 = Lamp is off
- Lamp 4: +5 V on Wire4 = Lamp is on; 0 V on Wire4 = Lamp is off

CHAPTER 2: STATE OF THE ART

It is easy to see that we need one wire per lamp.

With a bus system:

- Lamp 1: +1 V on Wire1 (Selector); 0 or +5 V on Wire2 for on and off (switch)
- Lamp 2: +2 V on Wire1 (Selector); 0 or +5 V on Wire2 for on and off (switch)
- Lamp 3: +3 V on Wire1 (Selector), 0 or +5 V on Wire2 for on and off (switch)
- Lamp 4: +4 V on Wire1 (Selector), 0 or +5 V on Wire2 for on and off (switch)

With this primitive kind of bus system we reduced the amount of wires to two. Regardless of the number of lamps we like to control, we only need one wire to tell the other chip which lamp we like to switch and a second wire to tell it whether we like to have the lamp on or off. Nevertheless, this example would have limits in the real world as one cannot simply raise the voltage to 1000 V on Wire 1 to switch a thousand different lamps.

But this example shows why in electronics in general, and in cars in particular, bus systems are being used. In the automotive industry cars use a number of bus systems that were made especially for them. Usually the features that matter the most are: [1]

- Reliability
- Low cost
- Maximum delay time
- Safety

Also, depending on the information is wanted to share, manufacturers choose a net like:

- CAN
- MOST
- FlexRay
- K-Line

For example, for the transmission of HQ video it is usually MOST because it has plastic fibre optic. However, if we would like to receive data with the maximum reliability we would decide to use FlexRay instead. It is convenient to say that CAN (controller area network) is the most important bus system in a car. We will not go into detail on this point, but simply just think of it as a way to transfer big amounts of data using only two wires.

In case this is not clear, it is also remarkable to distinguish the difference between an OBD-II protocol and a CAN setup. **OBD-II** is a **higher-level protocol** used for diagnostic purposes. OBD-II can use one of (many) different bus systems to transfer diagnostic data from and to the car. Think of OBD-II as a **code**, for example like the language (English) we speak, and of **CAN bus system** as the **communication device**, for example the telephone we use to talk to someone (about our car and its state of health).

Many people are referring to OBD (short for on-board diagnosis) or OBD-II as "standards". OBD-II is a standard, but it again consists of so many different standards, protocols and bus systems used to communicate that it is difficult to list all of them.

2.2 PROTOCOLS

2.2.1 CAN Bus Protocol

CAN Bus is a protocol developed by the German company Bosch to transmit messages between different environments. Initially this protocol was created to be used in vehicles and for this reason the platform features are the result of the needs in the automotive sector. In order to transmit the information, this bus breaks it down in messages and an only identifier is given to each message. Then the different nodes decide if accepting/refusing the message according to the identifier. It offers different benefits such as:

- Immunity to interference, ability for auto diagnostics and fix data errors.
- It is normalized so it makes easier to communicate systems made by different manufacturers.
- Multiplexed net without so many cables.

Moreover, there are two different CAN nets. One works with a higher speed (1 Mbit/s) and is used to monitor the engine and interconnect the ECU. The other one is used to communicate the rest of the parts of the vehicle such as doors, seats or lights and works with less speed (250 Kbit/s). In addition, there are four kinds of CAN frame:

-Data frame: The CAN data frame also works with two different protocols. The first one is called “base format” and has an identifier of 11 bits. The second one is the “extended format” and the identifier has 29 bits. The standard says that a CAN controller must accept at least basic frames but can (not) accept extended frames.

-Remote frame: Essentially it works the same as the previous one but there is a difference. It is possible that a node requires some data from another one. Then, a remote frame is requested to the second one in order to get the information. Basically, the difference between data frames and remote frames is that the last ones do not have data field.

-Error frame: This is a special frame that is transmitted when a node detects a wrong message. Then, the rest of nodes also transmit an error frame. There is an error counter that avoids the blockade of the bus with continuous errors.

-Overload frame: It is similar to error frame. It is transmitted by a node when it is very busy. Then the bus start providing extra delays between the nodes.

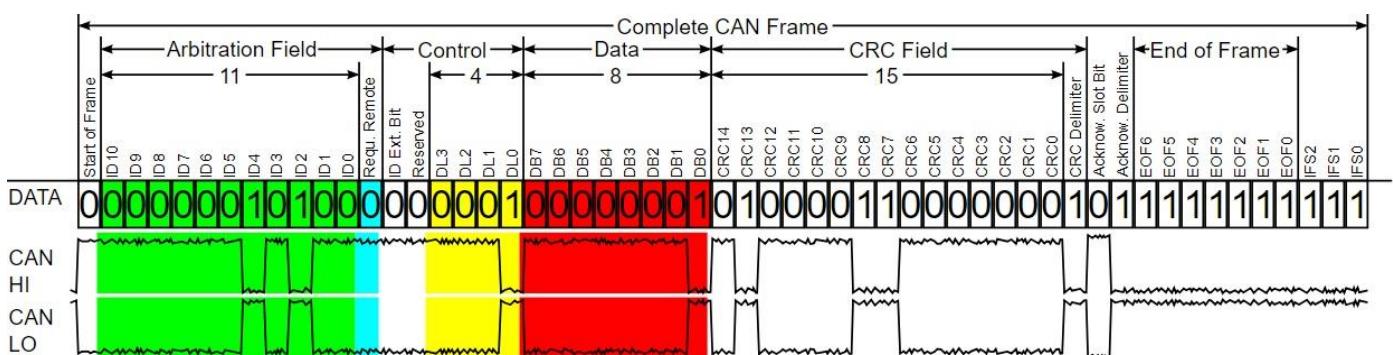


Figure 2.2.1a - Simplified CAN-Frame in base format [2]

Now, in the table below let us see what each bit is dedicated to.

Field Name	Length (bits)	Purpose
Start of frame	1	Denotes the start of frame transmission
Identifier (green)	11	A (unique) identifier which also represents the message priority
Remote Transmission Request (RTR) (blue)	1	Must be dominant (0) for data frames and recessive (1) for remote request frames
Identifier Extension Bit (IDE)	1	Must be dominant (0) for base frame format with 11-bit identifiers
Reserved bit (r0)	1	Reserved bit. Must be dominant (0) but accepted as either dominant or recessive.
Data length code (DLC) (yellow)	4	Number of bytes of data (0–8 bytes)
Data Field (red)	0–64 (or 0–8)	Data to be transmitted (length in bytes dictated by DLC field)
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End of Frame (EOF)	7	Must be recessive (1)

Table 2.2.1 - CAN-Frame in base format [3]

Finally let us appreciate an example of how the transfer layer works:

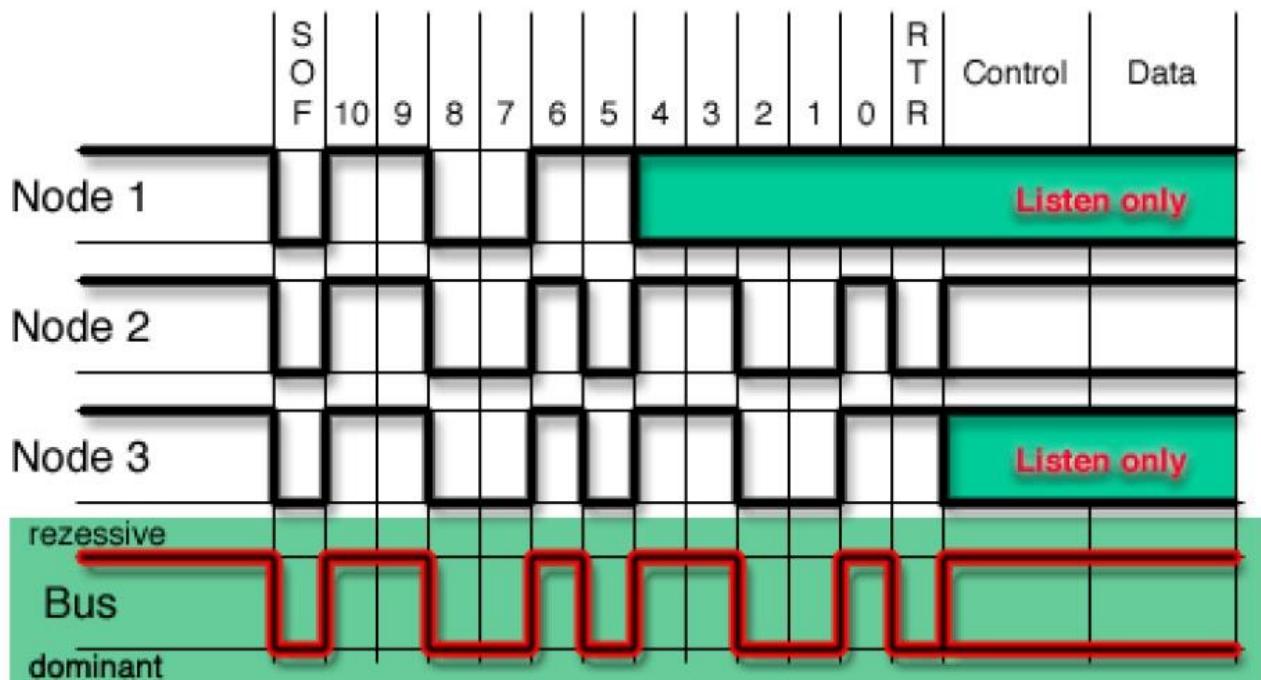


Figure 2.2.1b - CAN BUS transfer layer

In the example we can see a transmission from three nodes at the same time.

NODE 1: 11001100000

NODE 2: 11001011001

NODE 3: 11001011001

There are three bits involved in the arbitration: the initial bit, another one for the identifier and the last one just in case it is for remote transmission request (RTR). Here in the example, when the node 1 detects the 6th bit is dominant (0) but is not corresponding with the transmitted (1) it stops sending and will not send any message until 6 bit cycles after. That permits CAN bus to avoid delays in priority messages. Then, there is a special situation between nodes 2 and 3 due to that they have the same identifier. This should never happen because the same identifier is not given to different messages. However, in this trivial case we can see that node 3 is really doing a request (RTR bit is activated). Of course, RTR bit is recessive and the transmitted message will have the priority. Once there is no node detected from RTR bit, transmission will continue normally.

2.2.2 MOST Protocol

The Media Oriented Systems Transport (MOST) protocol is designed for multimedia devices. Typically, MOST is laid out in a ring topology, or virtual star, that supports a maximum of 64 MOST devices. One MOST device acts as the timing master, which continuously feeds frames into the ring. MOST runs at approximately 23 Mbaud and supports up to 15 uncompressed CD quality audio or MPEG1 audio/video channels. A separate control channel runs at 768 Kbaud and sends configuration messages to the MOST devices.

MOST comes in three speeds: MOST25, MOST50, and MOST150. Standard MOST, or MOST25, runs on plastic optical fiber (POF). Transmission is done through the red light wavelength at 650 nm using an LED. A similar protocol, MOST50, doubles the bandwidth and increases the frame length to 1025 bits. MOST50 traffic is usually transported on unshielded twisted-pair (UTP) cables instead of optical fiber. This one also has a similar layout to MOST25 but with a larger data section. Finally, MOST150 implements Isochronous and Ethernet so it increases the frame rate to 3072 bits or 150Mbps—approximately six times the bandwidth of MOST25.

Each MOST frame has three channels: Synchronous, Asynchronous and Control. In addition to a timing master, a MOST network master automatically assigns addresses to devices, which allows for a kind of plug-and-play structure. Another unique feature of MOST is that, unlike other buses, it routes packets through separate import and export ports. [4]

2.2.3 FlexRay Bus

FlexRay is a high-speed bus that can communicate at speeds of up to 10Mbps. It is geared for time-sensitive communication, such as drive-by-wire, steer-by-wire or brake-by-wire. FlexRay is more expensive to implement than CAN, so it is common that most implementations use FlexRay for high-end systems, CAN for midrange, and LIN for low-cost devices.

FlexRay supports a standard bus topology, like CAN bus, where many ECUs run off a twisted-pair bus. It also supports star topology, like Ethernet, that can run longer segments. When implemented in the star topology, a FlexRay hub is a central, active FlexRay device that talks to the other nodes. In a bus layout, FlexRay requires proper resistor termination, as in a standard CAN bus. The bus and star topologies can be combined to create a hybrid layout if desired.

CHAPTER 2: STATE OF THE ART

In a bus layout, only one device can talk on the bus at a time. The FlexRay Bus uses something called a *time division multiple access* (TDMA) scheme to guarantee the order by determinism: the rate is always the same (deterministic), and the system relies on the transmitters to fill in the data as the packets pass down the wire, similar to the way cellular networks like GSM operate. FlexRay devices don not automatically detect the network or addresses on the network, so they must have that information programmed in at manufacturing time.

FIBEX is an XML format used to describe FlexRay, as well as CAN, LIN, and MOST network setups. FIBEX topology maps record the ECUs and how they are connected via channels, and they can implement gateways to determine the routing behavior between buses. These maps can also include all the signals and how they are meant to be interpreted. FIBEX data is used during firmware compile time and allows developers to reference the known network signals in their code. [5]

2.2.4 SAE J1850

The SAE J1850 protocol was originally adopted in 1994 and can still be found in some of today's vehicles, for example some General Motors and Chrysler vehicles. These bus systems are older and slower than CAN but cheaper to implement. There are two types of J1850 protocols: pulse width modulation (PWM), typically used by Ford, and variable pulse width (VPW), used by General Motors and Chrysler. Also, the speed is grouped into three classes: A, B, and C.

The 10.4Kb/s speeds of PWM and VPW are considered class A, which means they are devices marketed exclusively for use in business, industrial, and commercial environments. Class B devices are marketed for use anywhere, including residential environments and have a second SAE standard implementation that can communicate at 100Kb/s, but it is slightly more expensive. The final implementation can operate at up to 1Mb/s, and it is used in class C devices. As it is expected, this third implementation is the most expensive, and it is used primarily in real-time critical systems and media networks. [6]

2.2.5 The OBD-III Standard

OBD-II was originally designed to be compliant with emissions testing, but now that the powertrain control module (PCM) knows whether a vehicle is within guidelines, the inconvenience of the vehicle owner having to go for testing still exists. However, the OBD-III standard allows the PCM to communicate its status remotely without the owner's interaction. This communication is typically accomplished through a roadside transponder, although cell phones and satellite communications work as well. The idea is to have the system report that pollutants are entering the atmosphere without having to wait up to two years for an emissions check. Nowadays, this system has some obvious legal questions that still need to be answered, including the risk of mass surveillance of private property. It is important to note that even if OBD-III sends only DTC and VIN, it is trivial to add additional metadata, such as location, time, and history of the vehicle passing the transponder. [7]

2.2.6 The Keyword Protocol

The Keyword Protocol 2000 (ISO 14230), also known as KWP2000, is common in American vehicles made after 2003. The messages sent using KWP2000 can contain up to 255 bytes. It works with pin 7 and has two different variations that differ mainly in baud initialization. ISO 9141-2, or K-Line, is the variation of KWP2000 seen most often in European vehicles. K-Line uses pin 7 and, optionally, pin 15. Unlike CAN packets, K-Line

packets have a source (transmitter) and a destination (receiver) address. Moreover, K-Line can use the same or a similar parameter ID (PID) request structure as CAN. [8]

2.2.7 The Local Interconnect Network Protocol

The Local Interconnect Network (LIN) is the cheapest of the vehicle protocols. It was designed to complement CAN. It has no arbitration or priority code; instead, a single master node does all the transmission. LIN can support up to 16 slave nodes that primarily just listen to the master node. A LIN message frame includes a header, which is always sent by the master, and a response section, which may be sent by master or slave. Often the LIN master node is connected to a CAN bus. The maximum speed of LIN is 20Kbps. LIN is a single-wire bus that operates at 12V. It is often used instead of direct CAN packets to handle controls to simple devices. [9]

3. AVAILABLE TOOLS

3.1 SOFTWARE

3.1.1 Delphi DS150

Combining versatility and ease of use with highly innovative functions and applications, Delphi's DS150 car software provides high level capability for an extensive range of brands and models. In fact, Delphi can be used in **more than 60 car and light commercial brands and over 82,000 vehicle systems**. [10]

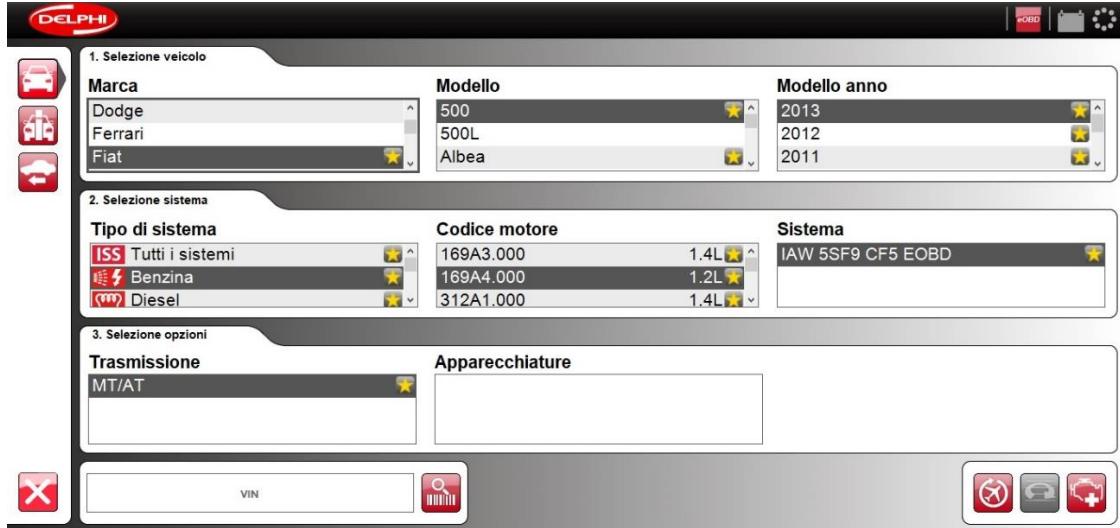


Figure 3.1.1a – Initial interface of Delphi

The software is available on a range of platforms including a convertible PC and tablets. With it, technicians can easily perform vehicle's behaviour, repair it or make some adjustments to the ECU. Many variables can be controlled or modified including petrol and diesel engine consumption, ABS, lights, Air Conditioned, gearbox, traction control, breaking system, etc...

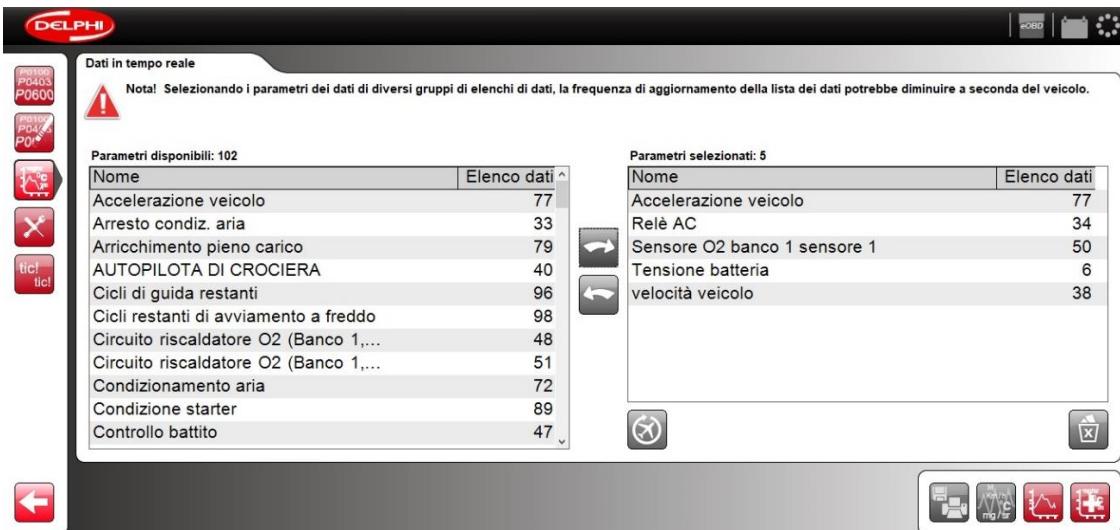


Figure 3.1.1b – PID's menu in Delphi

3.1.2 DashCommand

DashCommand™ is a touch screen friendly software application that is designed to integrate OBD-II data monitoring and logging into the in-car computing experience. Users can use its capabilities to create and display stunning virtual dashboards with styles ranging from digital gauges to analog gauges. If the need ever arises, this software is able to read and clear the trouble codes when the Check Engine Light comes on. DashCommand can also be used with multiple vehicles and helps users to keep track of each one using its built-in vehicle manager.

The software supports all OBD-II and EOBD compliant vehicles whether they are European, American or Asian. It is designed primarily for touchscreen devices but it is just as capable running on any desktop or laptop computer with a mouse or a touch pad. Although the app is available for both Android and iOS mobile phones, the computer version is compatible only with Windows (all versions). The iOS app has a price of 10.99€ while the Android version has a demo that allows us to use the full version but only for the first 30 minutes. DashCommand currently supports ELM compatible, Innovate Motorsports OT-2 and J2534 compliant OBD-II interfaces. Indeed, it supports all OBD-II protocols:

- SAE-J1850 (PWM and VPW)
- ISO-9141
- ISO-14230 (KWP2000)
- ISO-15765 (CAN)

This software counts with data logging capabilities to record logs from a dashboard or a data grid view and then playback the logs in either view for simple analysis tasks. Data can be exported to .csv format (and viewed as a table in excel). However, from more thorough analysis logs can also be viewed in ScanXL. This is a program developed by DashCommand to analyze tests data. Thanks to it, the basic data capabilities can be augmented with scripts written in ScanXL. The scripts can be imported or written to calculate fuel consumption, boost pressure, power, torque, and many others based on the OBD-II sensor values. [11]

3.1.3 EOBD Facile – Car Diagnostics

This software offers calculator for Diesel, gasoline, GPL and hybrid engine vehicles. It is able to create recordings including GPS data in .kml format. An advantage is that it is available to test/analyze from the smartphone using the app. The app can be downloaded easily from the Google Play/App Store because it is compatible with both Windows and Mac. Moreover, it can display specific manufacturer error codes for different manufacturers such as Renault, Peugeot, Citroën, Opel, BMW, Ford, Audi, Volkswagen, Skoda, Fiat, Alfa Romeo, etc... [12]

Nevertheless, this software requires an ELM327 interface to be connected into the OBD plug. In addition, it is working only with vehicles compatible with E OBD/OBD2 standard.

- EOBD (Europe), OBD2 (USA), JOBD (Japan)
- ISO 15765-4 (CAN 11bit/29bit 250/500 kb)
- ISO 14230-4 (KWP2000 slow and fast unit)
- ISO 9141-2
- J1850 (VPW & PWM)

CHAPTER 3: AVAILABLE TOOLS

Here we can see the oxygen test with the parameters & some other sensors.

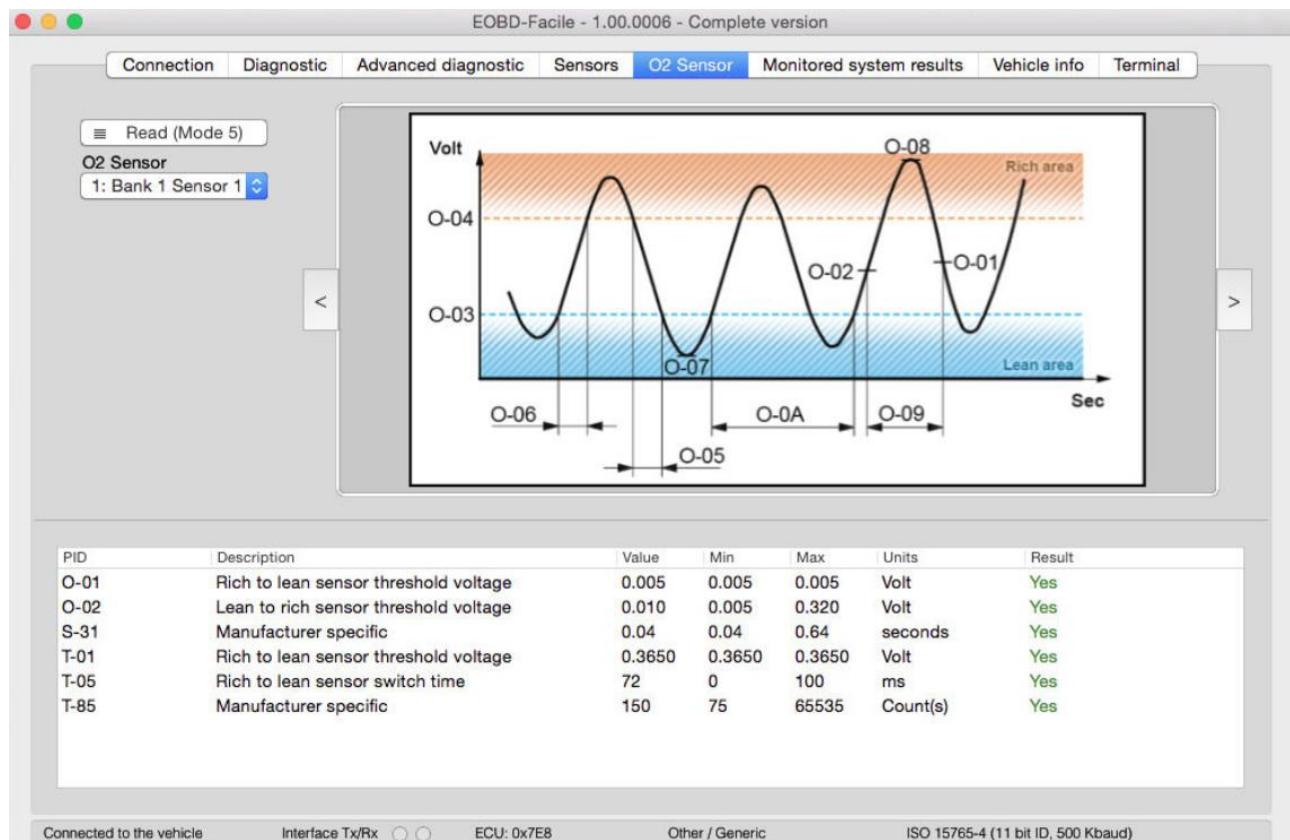


Figure 3.1.3a – Oxygen test with EOBD Facile [12]

As we see, we can see and print graphics with different parameters.

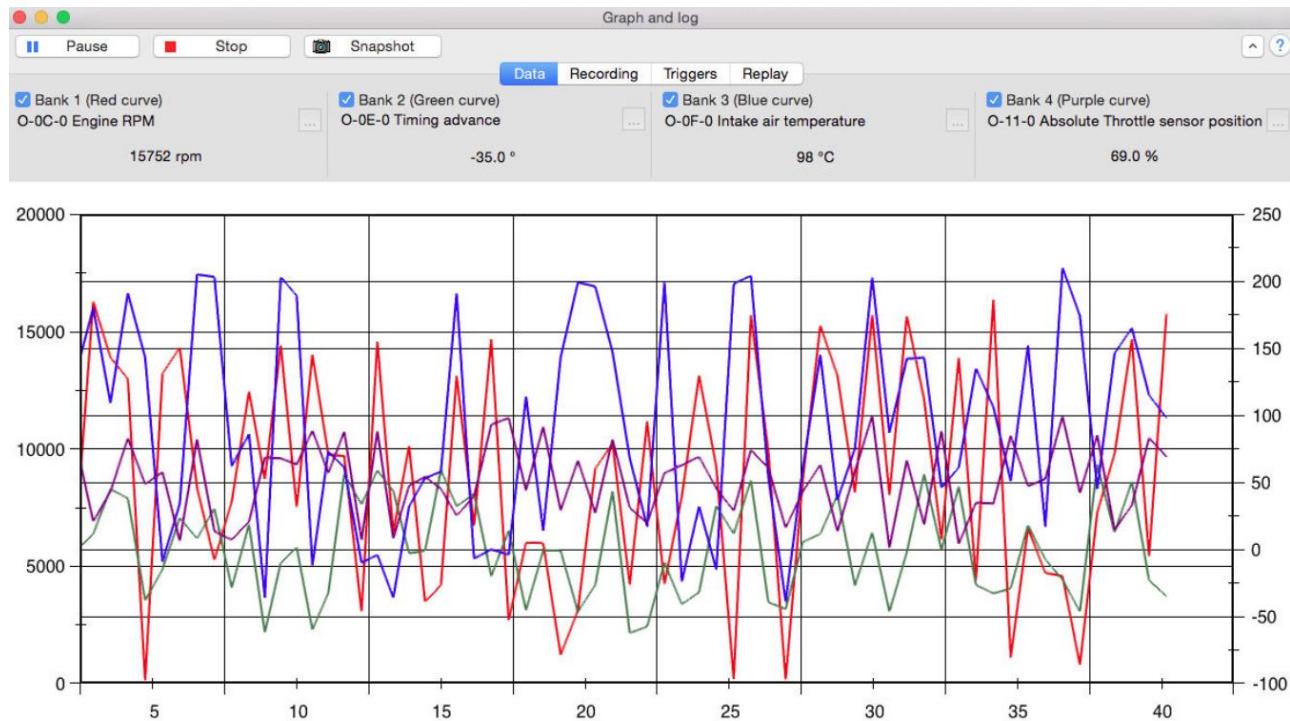


Figure 3.1.3b – Example of graphic with EOBD Facile [12]

3.1.4 OBD Auto Doctor

This software is especially indicated for those users that are not professionals who like cars and mechanics and use the cheapest version (50€) to check their own car using the app from the phone. But the full version includes many more options for a price of 120€. As well as the previous one, it has a fuel calculator (see image below) and allows us to export or save the live sensor data into a .csv file. Some other features are Freeze Frame, DTCs (Diagnostic Trouble Codes) & MIL.

The software can be bought on the company's website: <https://www.obdautodoctor.com/pricing-and-purchase> It is convenient to remark that an ELM327-based OBD-II adapter is needed to connect us to the vehicle (this is not included in the package).

About the communication standards we can say it works with:

- ISO 15765-4 (CAN) (required for advanced car diagnostics such as fuel system monitor)
- ISO 14230-4
- ISO 9141-2
- SAE J1850

Let us view a screenshot of the fuel consumption and the fuel emissions.

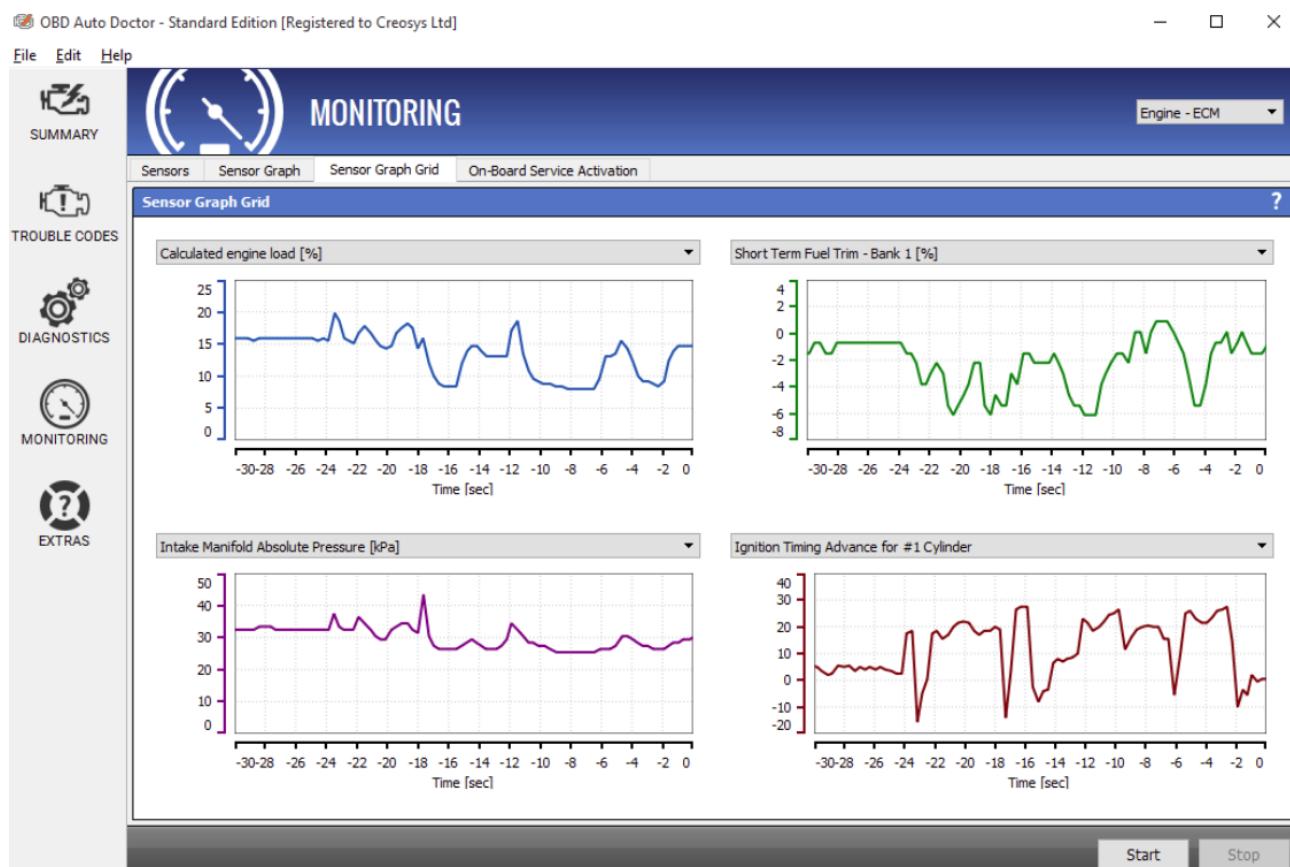


Figure 3.1.4a – Fuel consumption with OBD Auto Doctor [13]

CHAPTER 3: AVAILABLE TOOLS

Oxygen sensor monitoring test results include current test values as well as minimum and maximum values.

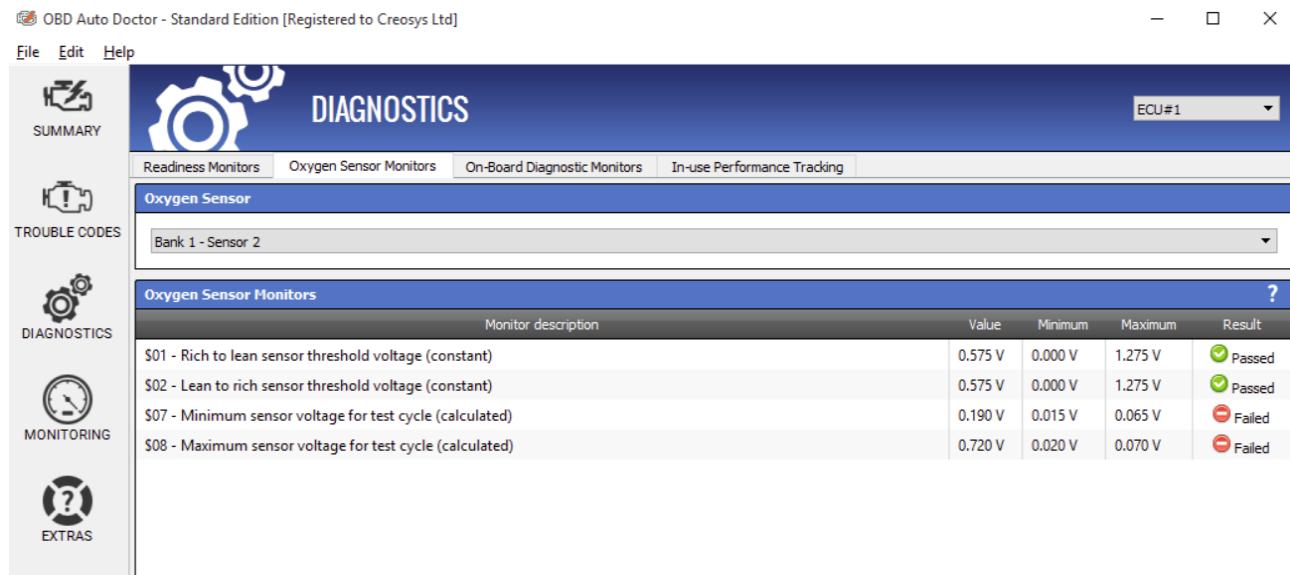


Figure 3.1.4b – Oxygen sensor appearance with OBD Auto Doctor [13]

The current version of the software allows us to pick up to 6 sensors to monitor simultaneously and plot them into a graph.

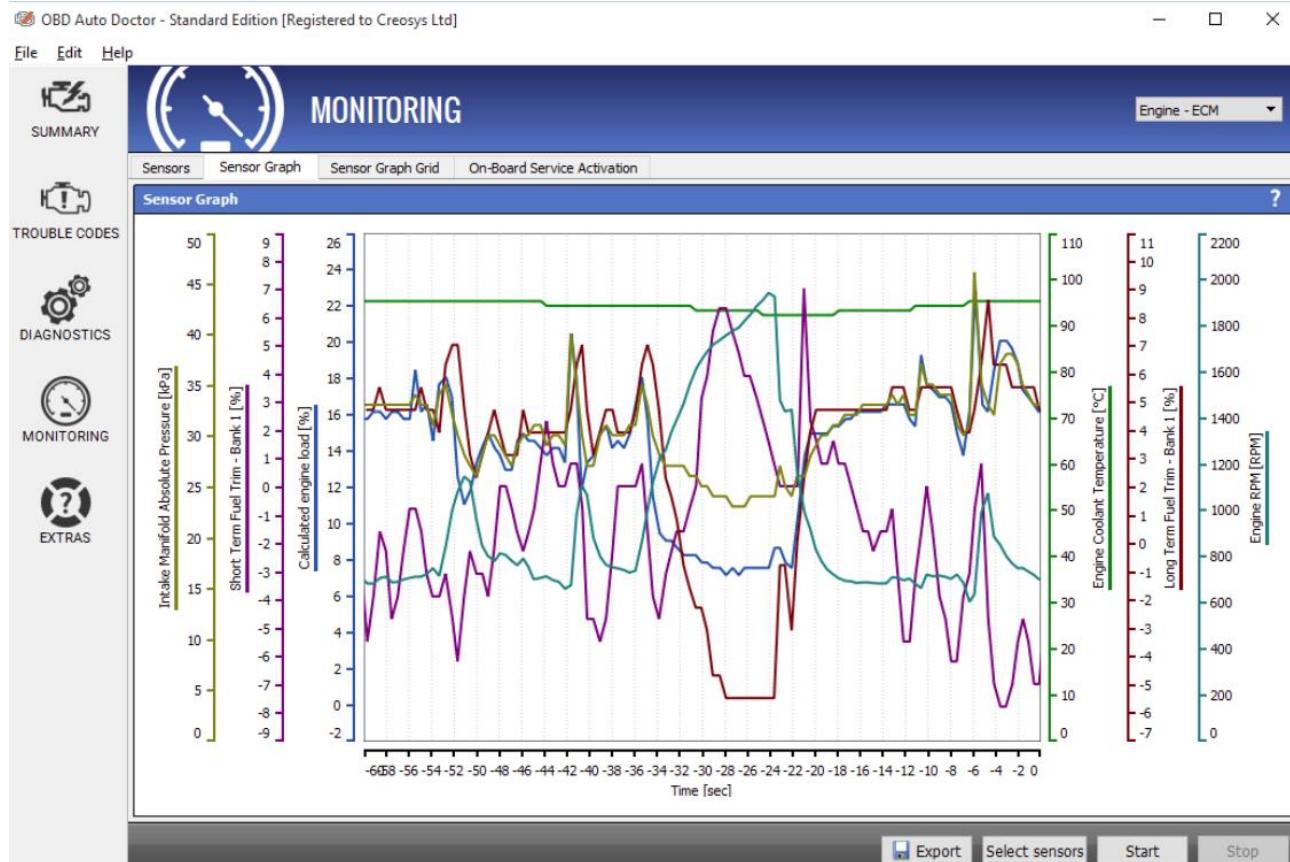


Figure 3.1.4c – Example of graph with multiple variables with OBD Auto Doctor [13]

CHAPTER 3: AVAILABLE TOOLS

This car diagnostic software supports built-in DTC database including 14000+ alarm codes.

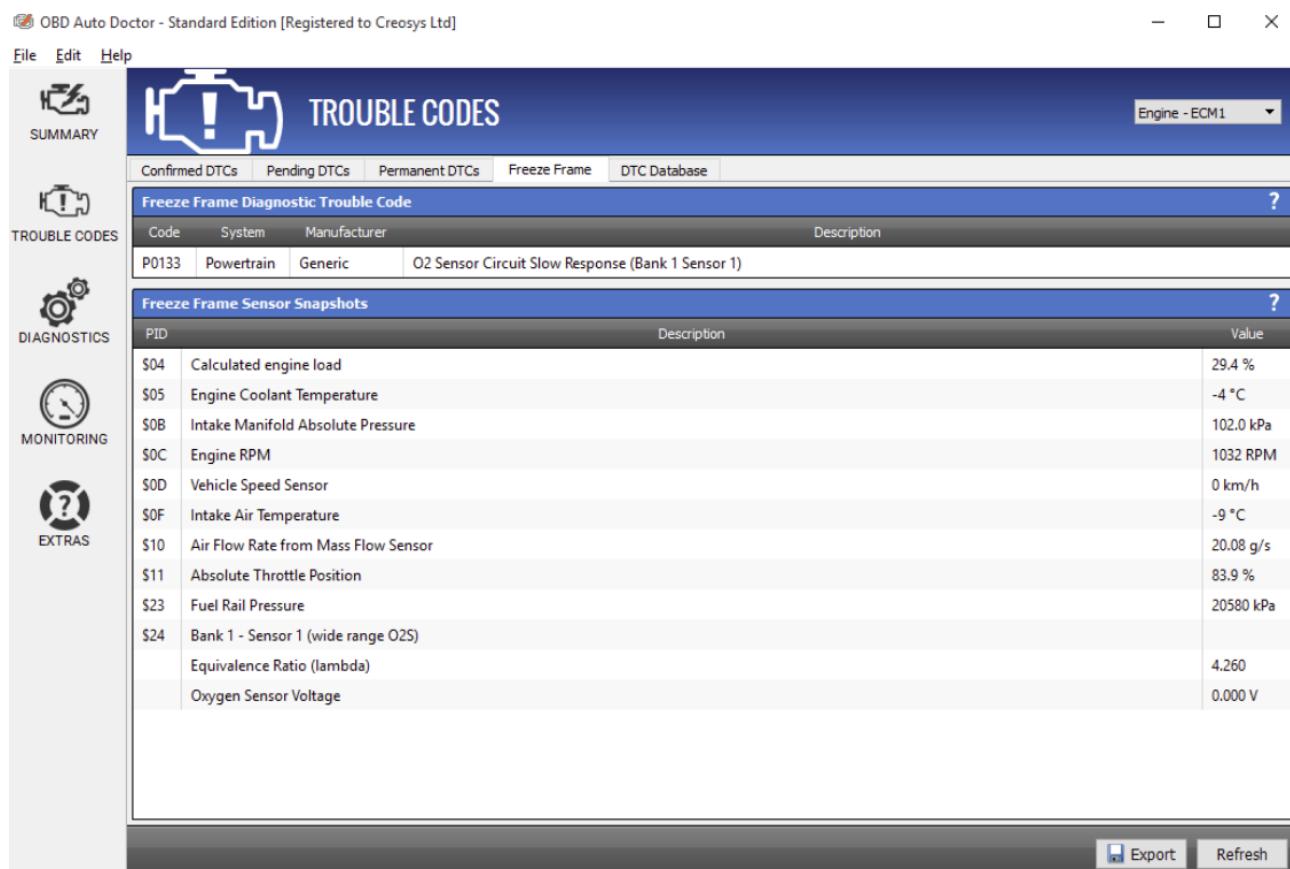


Figure 3.1.4d – Example of some PID's values with OBD Auto Doctor [13]

3.1.5 Movi Pro

This software offers two versions: users have to choose between the demo version (the demo is for free) and the full version that has a price of \$50. Both versions can be downloaded/purchased on the website <https://www.yhasi.com/support/software.php#registration>. It is remarkable that this software can only be installed in iOS' computers.

For connecting it is necessary to have any Bluetooth or USB interface that uses the ELM327 chip. It is convenient to inform that for RS232 interfaces an USB to RS232 adapter can be used. Movi Pro also provides the ability to connect using Wifi interfaces.

It can combine different parameters in order to create new ones. For example, we could view our vehicle's instantaneous fuel economy. If the vehicle returns the MAF air flow rate and speed through the OBD II system, Movi Pro can calculate the instantaneous fuel economy.

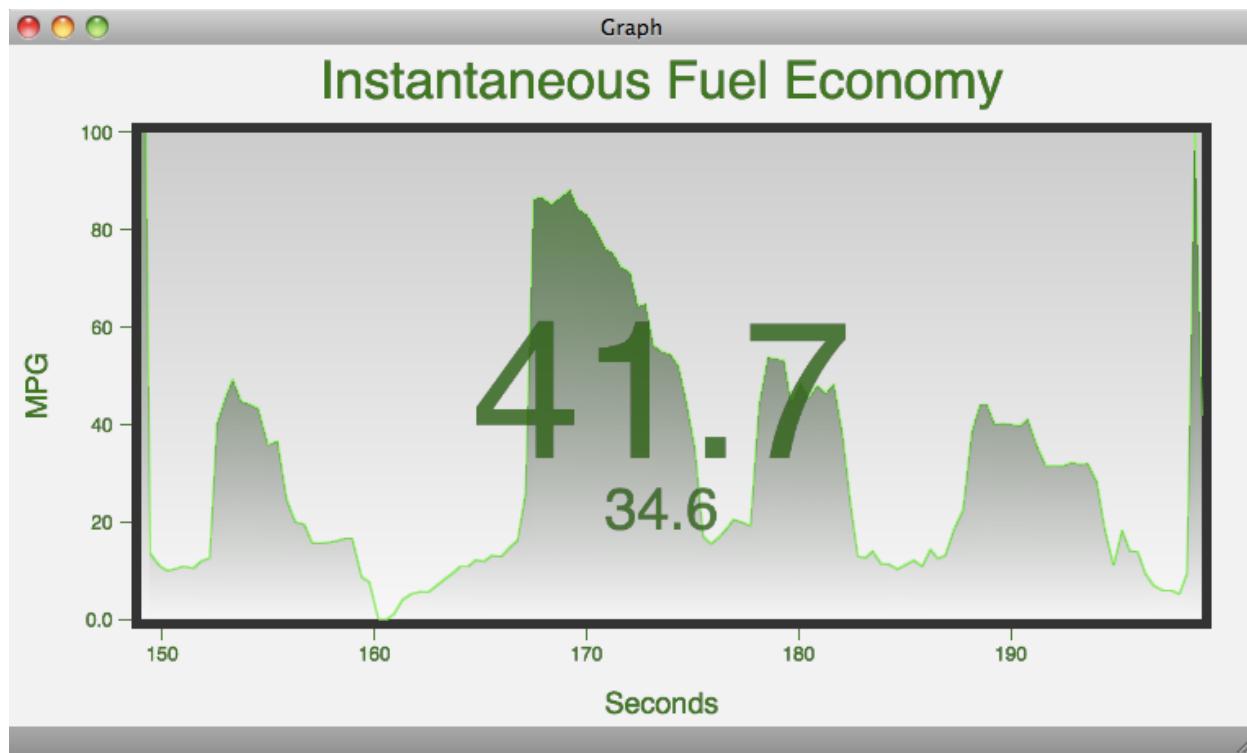


Figure 3.1.5a – Graph of instantaneous fuel consumption with Movi Pro [14]

The Movi Pro software displays the current state the vehicle was in when the diagnostic trouble code was set.

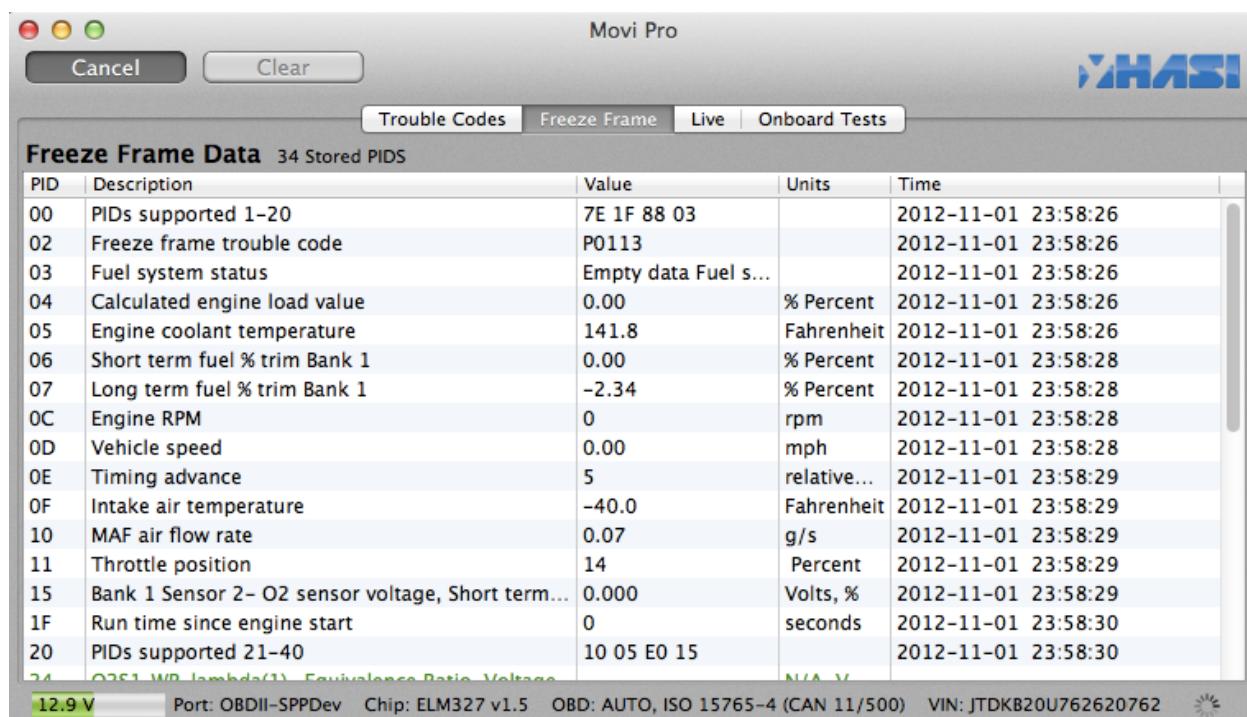
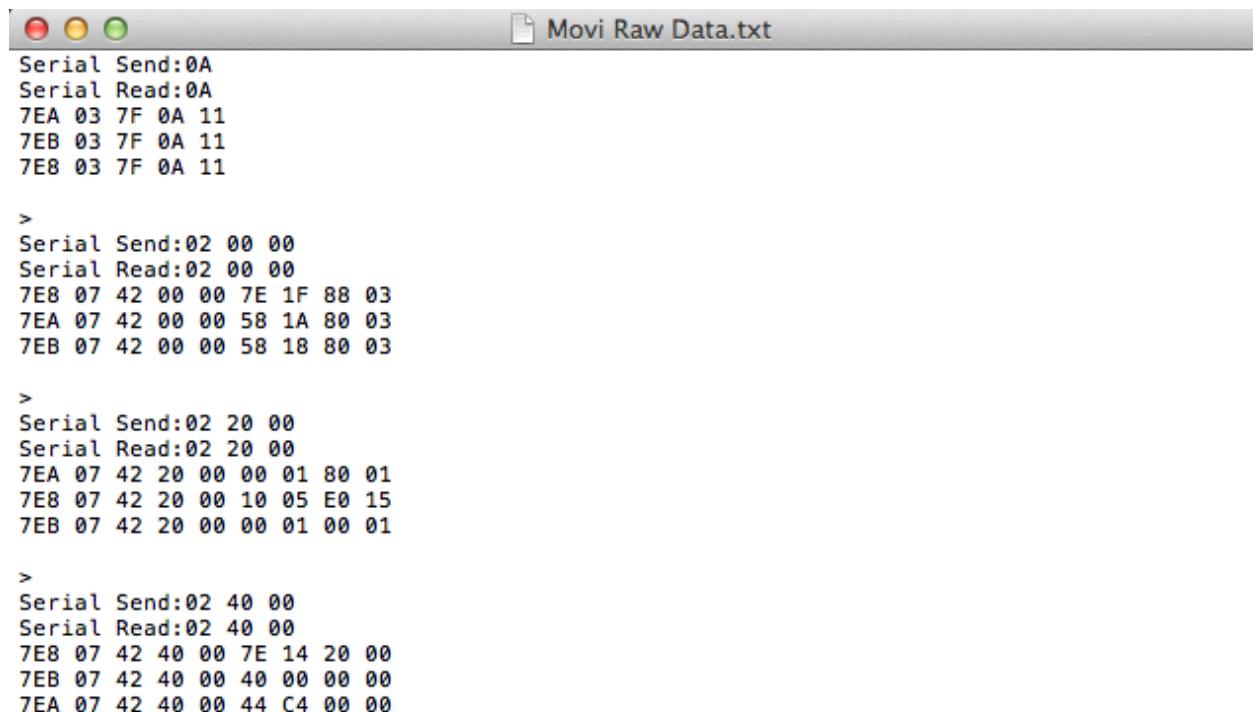


Figure 3.1.5b – List of PID's data with Movi Pro [14]

CHAPTER 3: AVAILABLE TOOLS

Users can view and save also the raw data from their vehicle's Engine Control Unit.



```
Serial Send:0A
Serial Read:0A
7EA 03 7F 0A 11
7EB 03 7F 0A 11
7E8 03 7F 0A 11

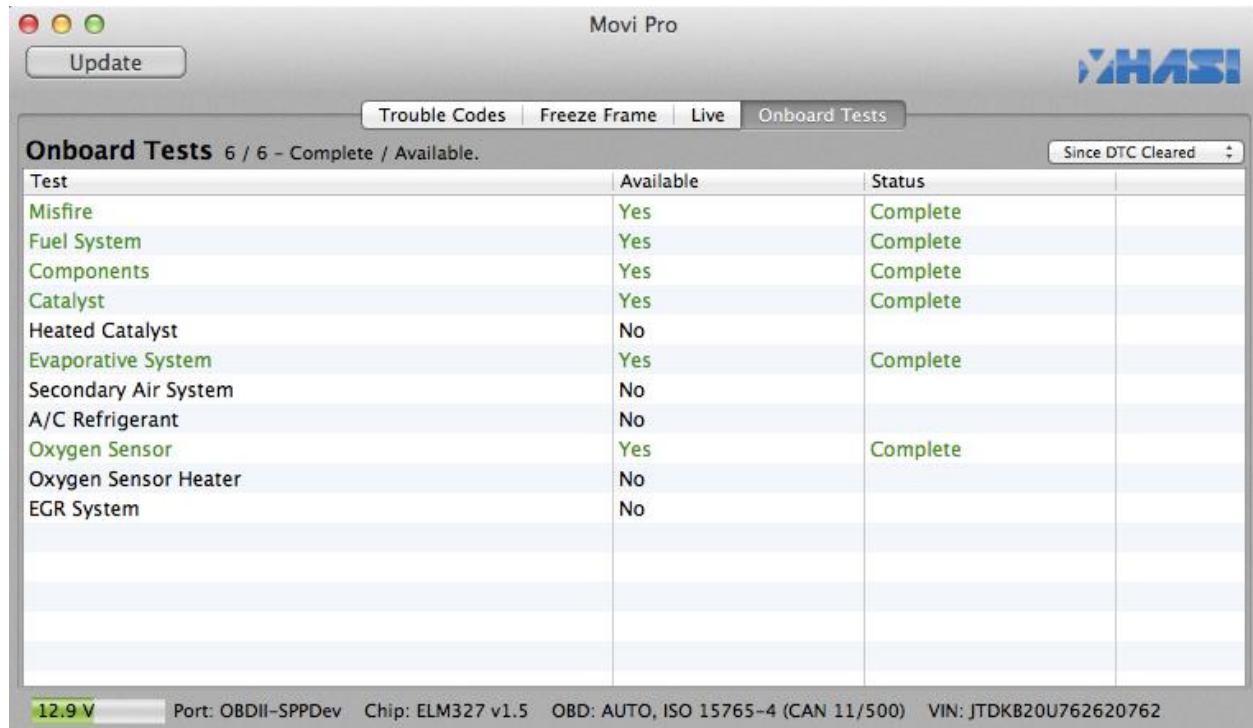
>
Serial Send:02 00 00
Serial Read:02 00 00
7E8 07 42 00 00 7E 1F 88 03
7EA 07 42 00 00 58 1A 80 03
7EB 07 42 00 00 58 18 80 03

>
Serial Send:02 20 00
Serial Read:02 20 00
7EA 07 42 20 00 00 01 80 01
7E8 07 42 20 00 10 05 E0 15
7EB 07 42 20 00 00 01 00 01

>
Serial Send:02 40 00
Serial Read:02 40 00
7E8 07 42 40 00 7E 14 20 00
7EB 07 42 40 00 40 00 00 00
7EA 07 42 40 00 44 C4 00 00
```

Figure 3.1.5c – Raw data with Movi Pro [14]

Movi Pro displays the onboard tests our vehicle's engine control unit performs every drive cycle.



The screenshot shows the Movi Pro software interface. At the top, there are three window control buttons (red, yellow, green) and a title bar with the text "Movi Pro". Below the title bar is a toolbar with a "Update" button and the "MAHSAI" logo. The main window has a tab bar with "Trouble Codes", "Freeze Frame", "Live", and "Onboard Tests" selected. A status message "Onboard Tests 6 / 6 - Complete / Available." is displayed above a table. The table has columns for "Test", "Available", and "Status". The rows list various vehicle systems: Misfire, Fuel System, Components, Catalyst, Heated Catalyst, Evaporative System, Secondary Air System, A/C Refrigerant, Oxygen Sensor, Oxygen Sensor Heater, and EGR System. Most systems are marked as "Complete" and "Yes" for availability, except for the Secondary Air System which is marked as "No". At the bottom of the interface, there is a footer bar with the text "12.9 V", "Port: OBDII-SPPDev", "Chip: ELM327 v1.5", "OBD: AUTO, ISO 15765-4 (CAN 11/500)", and "VIN: JTDKB20U762620762".

Test	Available	Status
Misfire	Yes	Complete
Fuel System	Yes	Complete
Components	Yes	Complete
Catalyst	Yes	Complete
Heated Catalyst	No	
Evaporative System	Yes	Complete
Secondary Air System	No	
A/C Refrigerant	No	
Oxygen Sensor	Yes	Complete
Oxygen Sensor Heater	No	
EGR System	No	

Figure 3.1.5d – List of different tests with Movi Pro [14]

CHAPTER 3: AVAILABLE TOOLS

The software can display up to four simultaneous graphs of live engine data, all while recording and listening to the displayed PID's in real time. Moreover, the previously recorded data can also be loaded into the graph for comparisons and review.

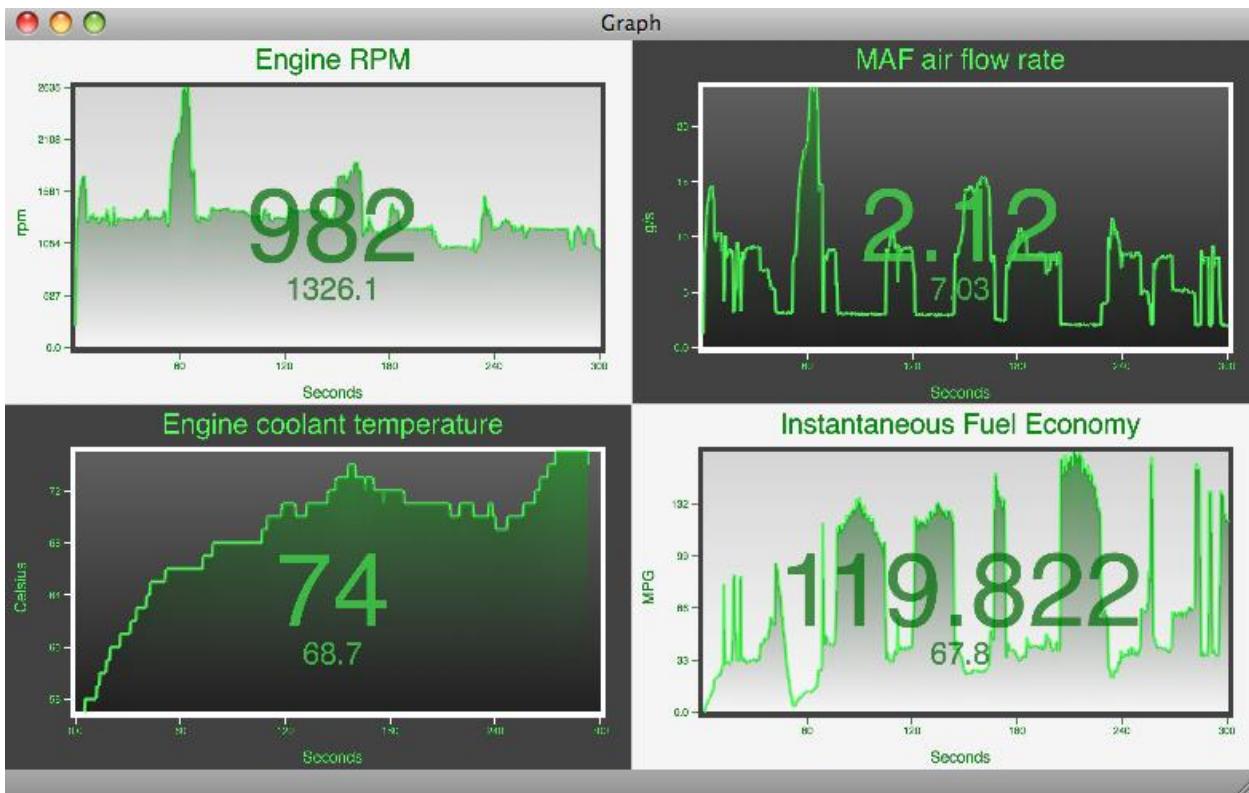


Figure 3.1.5e – Example of some live data graphics with Movi Pro [14]

We can see some graphs of monitoring of virtual torque and horsepower as an example of the simulations that can be done with this software.



Figure 3.1.5f – Example of a torque simulation with Movi Pro [14]



Figure 3.1.5g – Example of a horse power simulation with Movi Pro [14]

3.1.6 AutoEnginuity ScanTool

This software can be acquired on the website <https://www.autoenginuity.com/products/scantool/>. The price rounds \$250 and there is no demo or free version. However, once acquired the package all later updates are for free. It works with the majority of vehicles worldwide like Ford-family, GM-family, Toyota/Scion/Lexus, Chrysler & Dodge-family, Mazda, Nissan and Infiniti, BMW and MINI, Honda and Acura, Hyundai and Kia, Land Rover, Jaguar, Subaru, Porsche, Isuzu, Mitsubishi, Audi & Volkswagen, Mercedes, Ferrari & Maserati and Fiat.

About the connexion we can conclude it allows us to connect either with an OBD-II or with an USB 2.0 cable. Both hardware connectors come in the package when purchasing on the website. [15]



Figure 3.1.6a – Package of AutoEnginuity ScanTool [15]

Figure 3.1.6 b– Example of Can interface with AutoEnginuity ScanTool [15]

CHAPTER 3: AVAILABLE TOOLS

It works with vehicles compatible with SAE Interfaces:

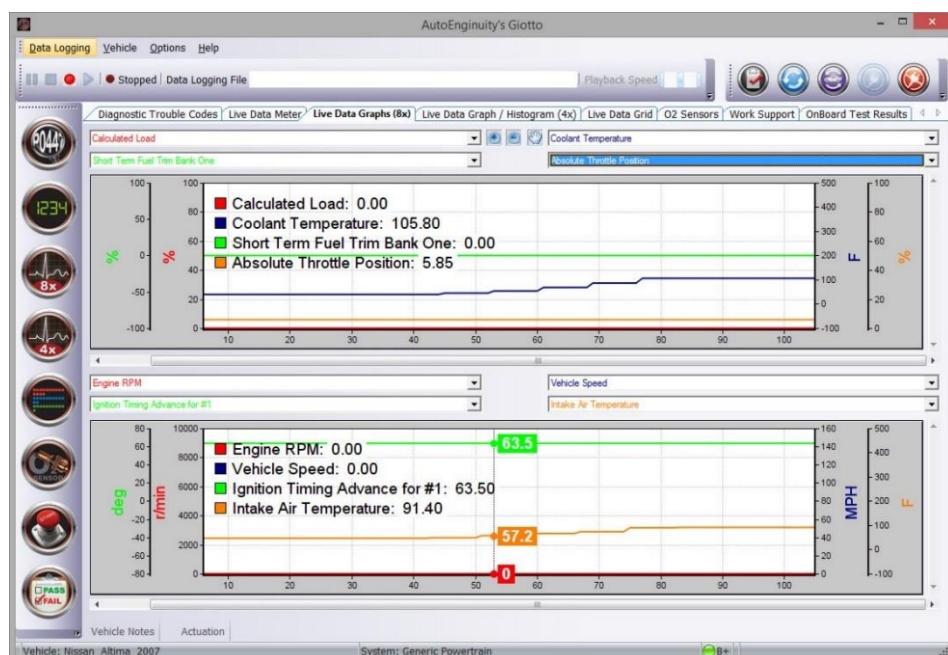
- CAN 11bit and 29bit
- ISO 14230 (KWP2000)
- ISO 9141-2
- J1850 (VPW & PWM)

Also when exporting the data we can log them in two different formats: XML for browsers and CVS for spread sheets with the capability to change the format and view logs offline. AutoEnginuity ScanTool software can read and clear Freeze Frame Data as we see in the next image.

STORED DTCs	
P1639	Vehicle ID Block Corrupted, Not Programmed
PENDING DTCs	
P701	Transmission Control System Range/Performance
P193	Fuel Rail Pressure Sensor Circuit High Input
P1639	Vehicle ID Block Corrupted, Not Programmed
P21	Intake Camshaft Position Timing - Over-Advanced (Bank 2)
P1000	OBD Systems Readiness Test Not Complete

FREEZE FRAME			
Freeze Frame DTC	P1639	Fuel System	Not Reported
Calculated Load	0.00 %	Coolant Temperature	-40.00 F
Short Term Fuel Trim Bank 1	-99.84 %	Long Term Fuel Trim Bank 1	-99.84 %
Short Term Fuel Trim Bank 2	-99.84 %	Long Term Fuel Trim Bank 2	-99.84 %
Fuel Pressure	Not Reported	Intake Manifold Pressure	Not Reported
Engine RPM	0.00 r/min	Vehicle Speed	0.00 MPH

Figure 3.1.6c – Freeze Frame Data with AutoEnginuity ScanTool [15]



The interface with the user is very comfortable when displaying data. Customers can change and personalize the interface that suits better of them. This software uses grids and graphics with different parameters and the 6 Data Mode can show 6 variables in an individual report format.

Figure 3.1.6d – Example of some graphics with AutoEnginuity ScanTool [15]

CHAPTER 3: AVAILABLE TOOLS

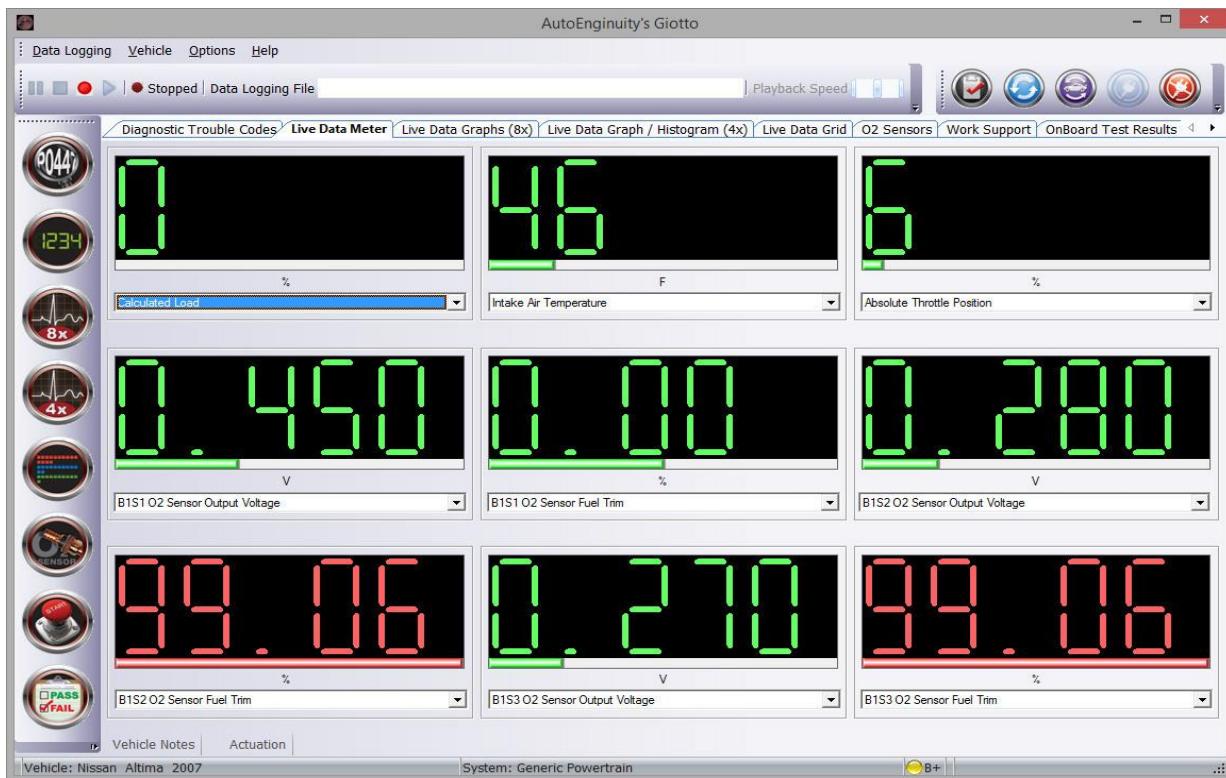


Figure 3.1.6e – 6 Data Mode with AutoEnginuity ScanTool [15]

Also sensor data can be configured. Each individual sensor's sampling rate, ranges, alert audio trigger points, scaling value or units can be set to the user's specific needs.

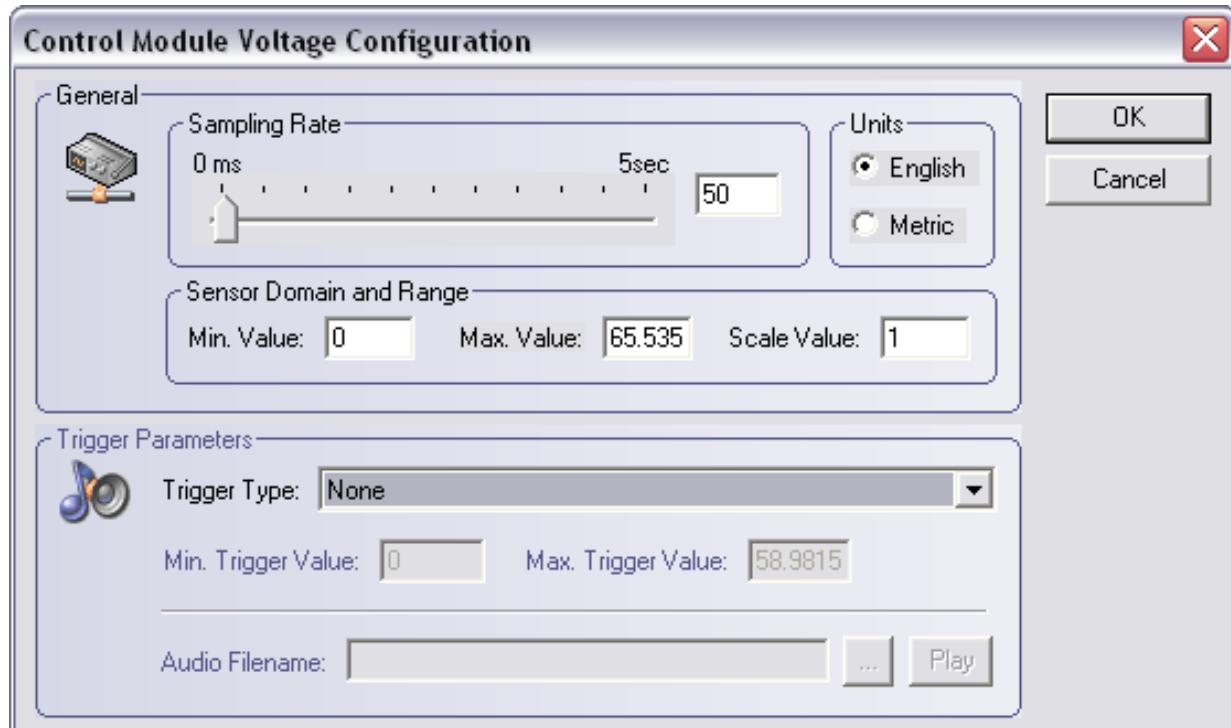


Figure 3.1.6f – Alert audio trigger with AutoEnginuity ScanTool [15]

CHAPTER 3: AVAILABLE TOOLS

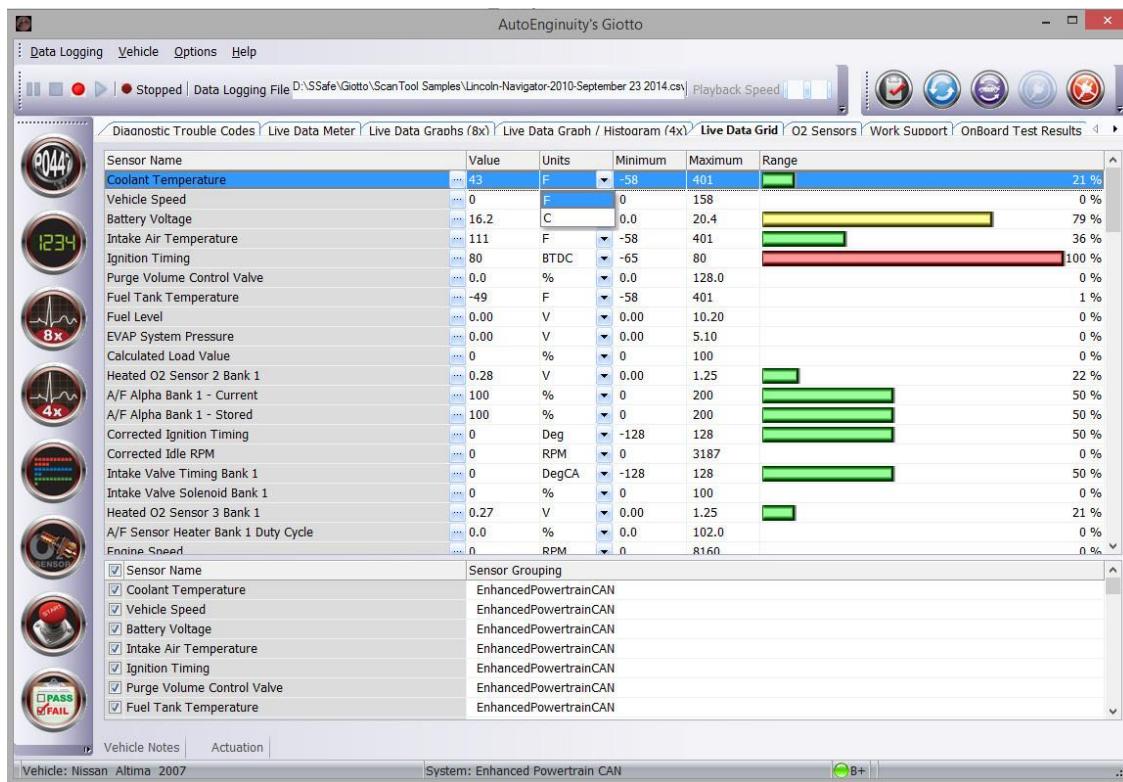


Figure 3.1.6g – Configuration of data sensor with AutoEnginuity ScanTool [15]

Oxygen sensor.

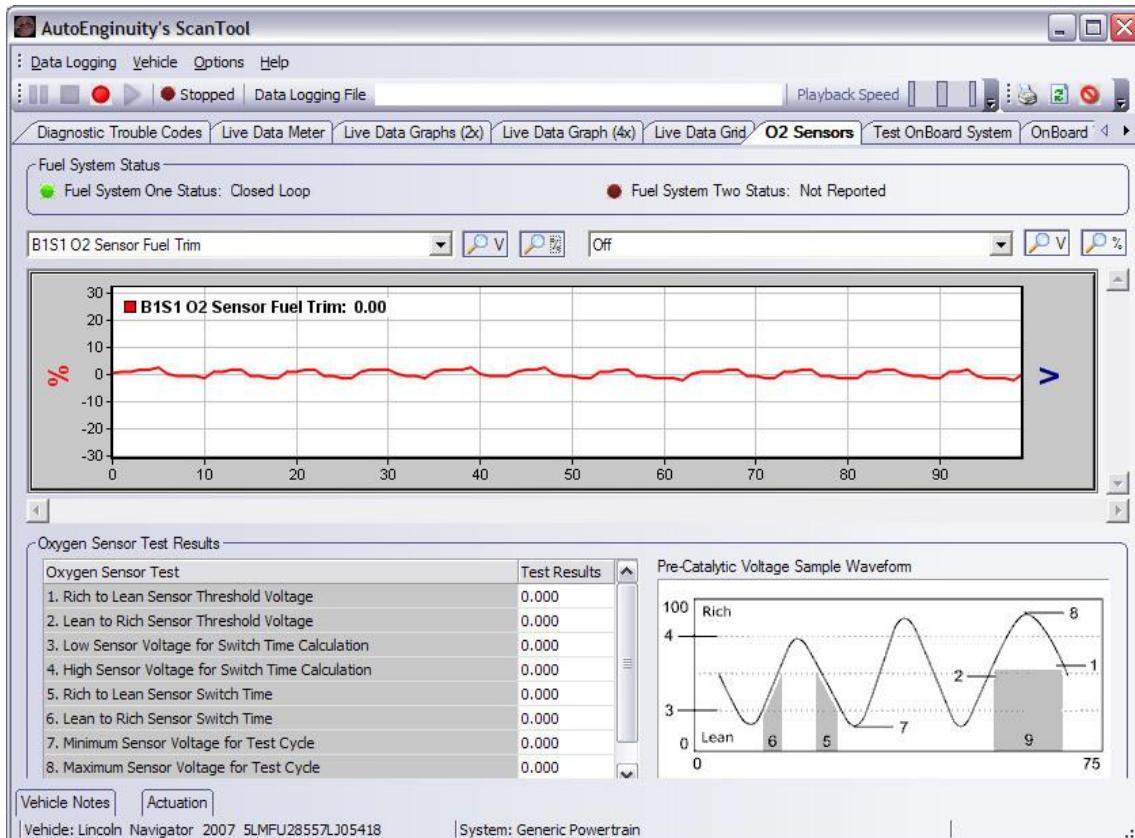


Figure 3.1.6h – Fuel sensor of AutoEnginuity ScanTool [15]

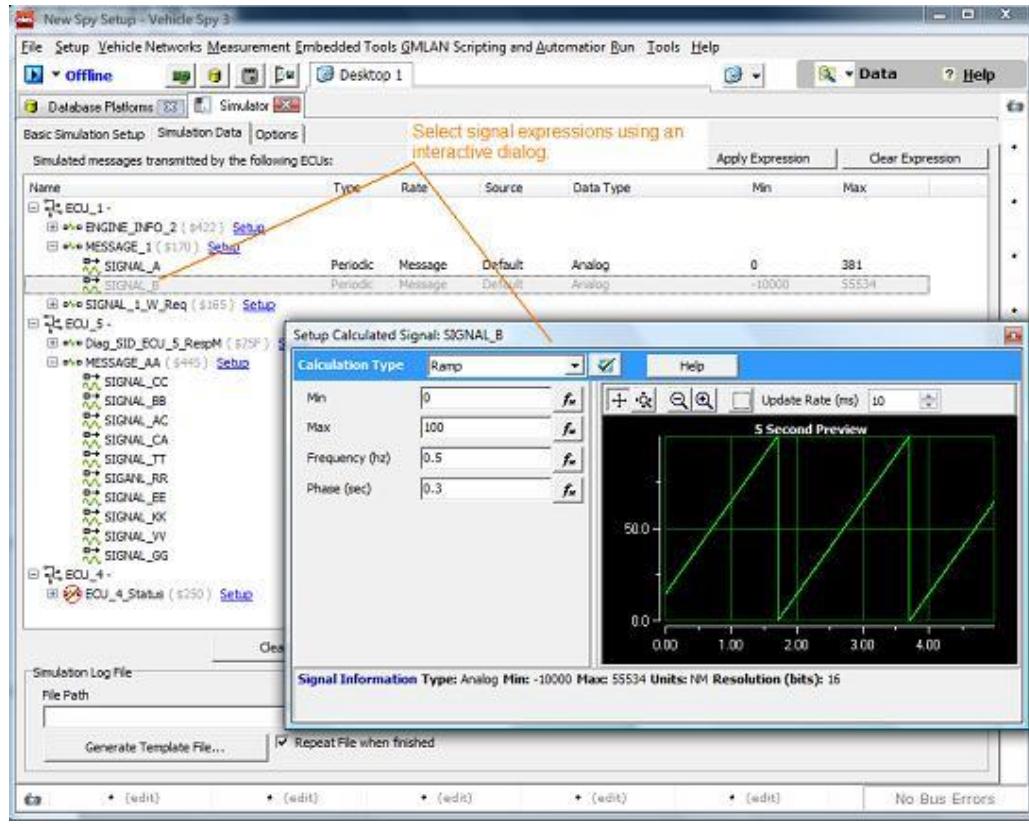
3.1.7 Vehicle Spy

Vehicle Spy is a software that can be bought for \$300 directly from the website. It works with vehicles compatible with the next communication standards:

- ISO 14229 (UDS)
- GMLAN
- CCP / XCP
- ISO 15765-2
- J1939
- J1979 (OBD)

Related with the exporting data format, industry standard .dbc and .ldf files are supported, as well as .uef and some other customer specific formats. Also, another important feature is the data log. It allows logging data with real-time signal views and post-analysis of data files provides user with a complete data acquisition and analysis solution.

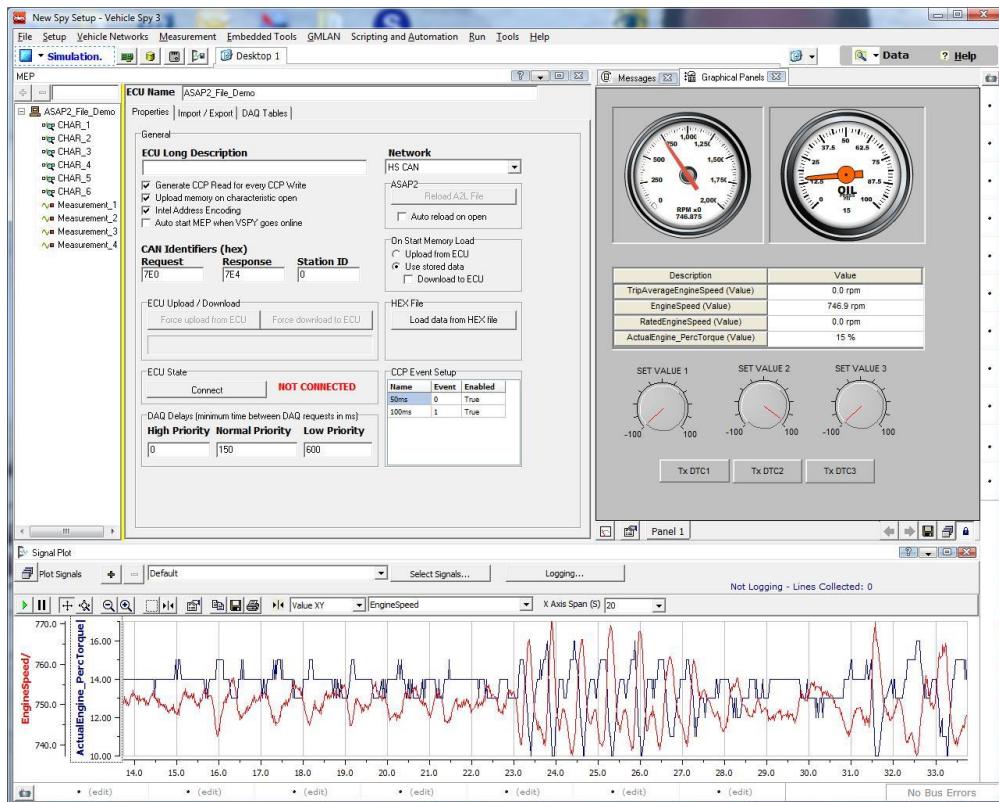
Vehicle Spy has an Applications Programmers Interface (API) that enables other programs to control its actions. This feature can be used by external programs on the same computer, or a remote computer with a TCP/IP connection. Programs written in LabView, C++, Visual Basic, MATLAB or other Windows programs are supported. [16]



Now let us see some features and advantages that this software has. For example, built-in Diagnostics Setup enables user to create and execute diagnostic jobs and save the results for later analysis.

Figure 3.1.7a – Diagnostic Setup of Vehicle Spy [17]

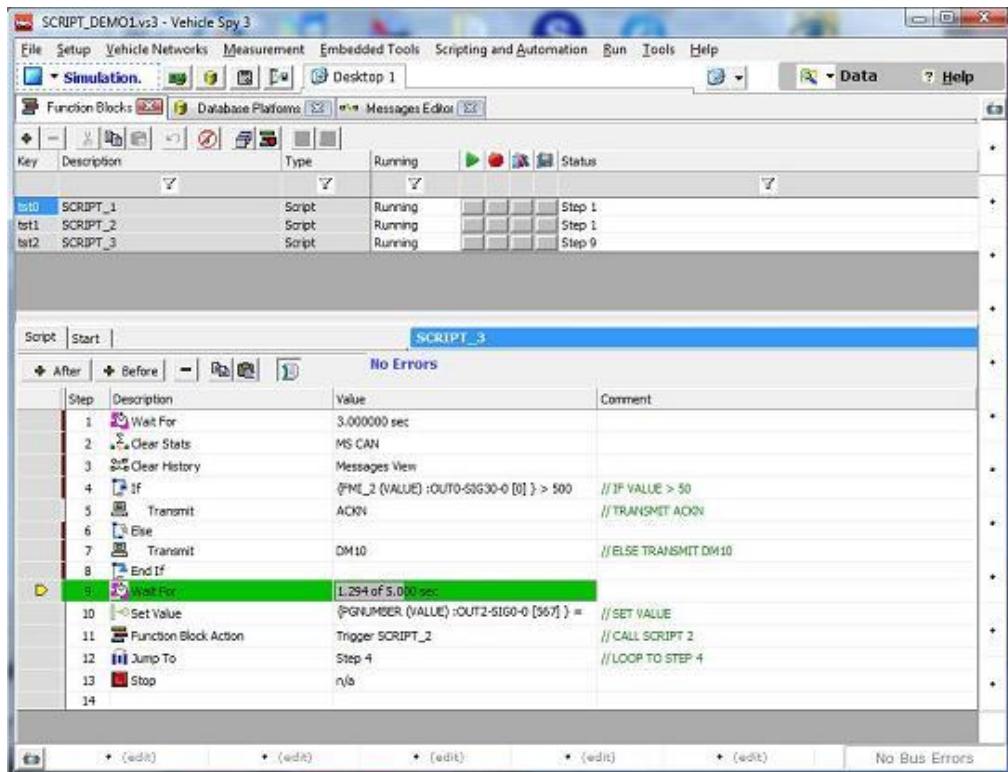
CHAPTER 3: AVAILABLE TOOLS



Our data logging capable devices enable users to set up standalone CCP/XCP data logging jobs. Normal mode CAN traffic can also be logged at the same time as CCP/XCP data. All data are automatically time-aligned and timestamped with a real-time clock.

Figure 3.1.7b – Data Logging with Vehicle Spy [17]

Vehicle Spy includes a built-in database and message editing facility. To create or edit messages user can go to the Messages Editor and then make the necessary changes to messages and signals. It is possible to create a custom user interface to view bus data the way user needs it, transmit messages, and interact with scripts and other parts of Vehicle Spy.

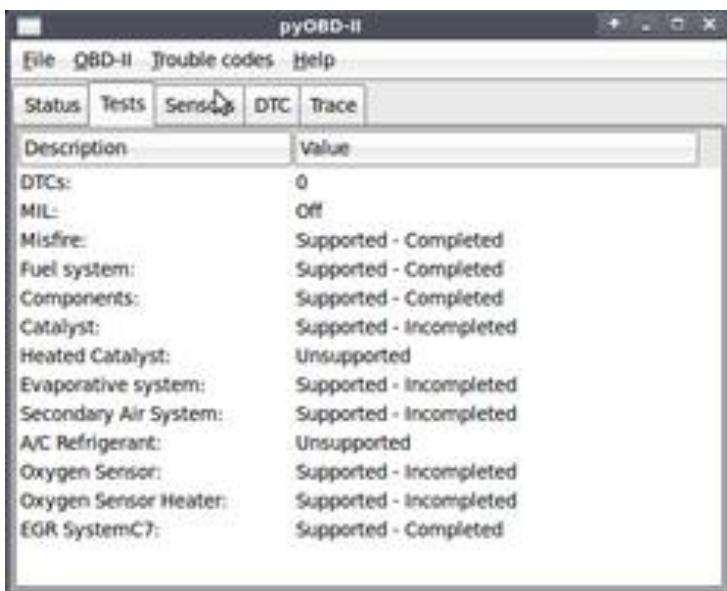


If we pay attention to the Function Block Scripting, we can see this is a graphical approach to scripting but allowing user to easily select a list of actions from a set of options for each step. This simplified approach is ideal for those whose main technical focus is not programming. Function Blocks offer the same power as other scripting languages but are much easier to learn and apply.

Figure 3.1.7c – Programming interface of Vehicle Spy [17]

3.1.8 PyOBD

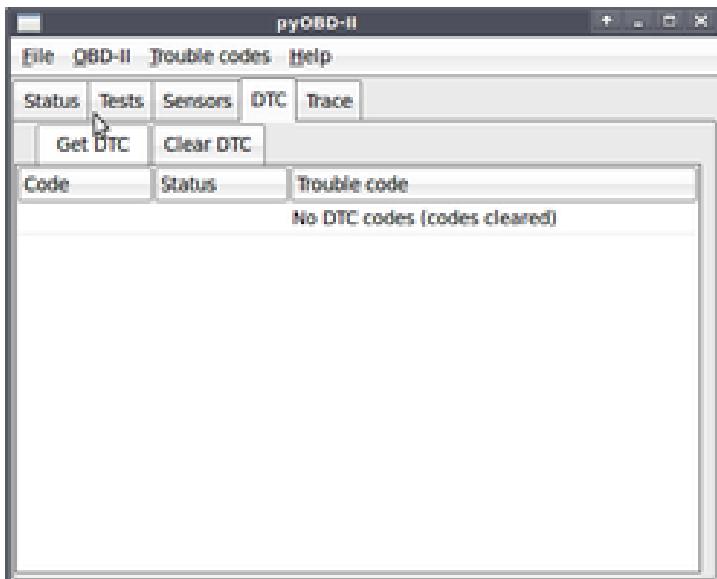
PyOBD (also pyOBD-II or pyOBD2) is a Czech OBD-II compliant car diagnostic tool. It is a Python module that was designed to be compatible with low-cost (ELM 323 or ELM327) OBD-II diagnostic interfaces such as ELM-USB. It can basically allow users to communicate with their car's ECU, display fault codes, display measured values, read status tests, etc. All cars made since 1996 (in the US) or 2001 (in the EU) must be OBD-II compliant so they should work with pyOBD. PyOBD is written entirely in Python and was originally written by Donour Sizemore, now maintained and improved by SECONS Ltd. and it is a free software and is distributed under the terms of the GPL. [18]



Freeze frame data is a snapshot of the real-time sensor feeds at the time of a DTC condition. Users can use these data to figure out what was going on at the time their car's "check engine" light turned on. OBD-II also provides detailed oxygen sensor data allowing emission diagnostics.

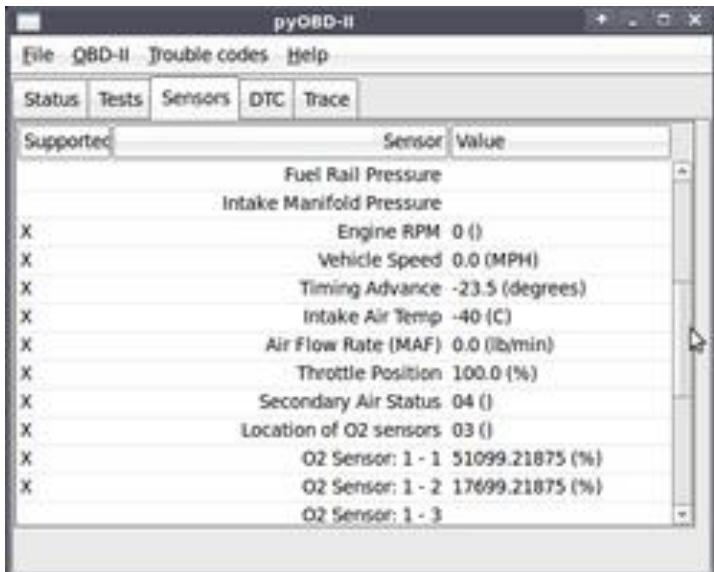
While the engine control unit (ECU) performs various tests, the OBD-II application allows users to view the results of them.

Figure 3.1.8b – Test results of PyOBD [18]



Diagnostic trouble codes (DTC) are error codes that can be looked up to determine what problem the car is experiencing. There are two types of fault memories: persistent (currently failing components) and non-volatile (stored errors that occurred in past). If the condition that caused the DTC persists, the car's computer will turn on the "check engine" light.

Figure 3.1.8c – Diagnostic Travel Codes (DTC) with PyOBD [18]



Real-time data are the sensor's raw data reported to the OBD-2 compliant control unit. These data can be helpful for troubleshooting problems and monitoring engine performance.

Figure 3.1.8d – Real time data sensor of PyOBD [18]

3.1.9 TOTAL OBD & ECU Auto Diagnostics (TOAD)

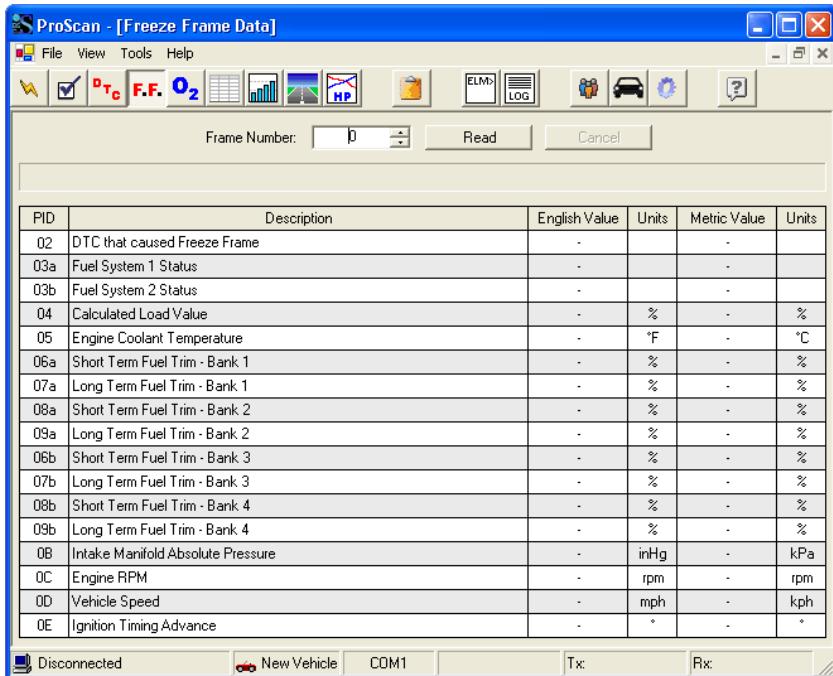
TOAD software can be purchased in Amazon or directly on the Internet in their website <http://www.totalcardiagnostics.com/toad/> for \$270. There are different connections available for OBD 1&2, EOBD, JOBD, ADR and JDM vehicles, the most common ones are ELM327 USB Cable / Bluetooth / WIFI. The software has a built-in database of over 15,000 fault code definitions (generic and manufacturer-specific codes for engine/transmission). It also works with vehicles compatible with the next communication standards:

- CAN Bus
- SAE-J1850
- CCP / XCP
- ISO-14230 and ISO-15765 (CAN)
- CAN [11bit and 29bit]
- ELM327

In comparison with other software, TOAD is able to read many more fault code DTC than another high-end reader and is also permitted to monitor multiple sensors in real time simultaneously. But another benefit is getting around 20% more data than competitors. It has widely been used by professional mechanics. It has been used on Toyota, BMW, Volkswagen, GM, Ford, Mercedes, Chrysler, Nissan, Skoda, Volvo and several more different customer vehicle manufacturers, and surprising TOAD can find PID's and additional emission status that usually only with high-end OBD readers (\$10,000+ range) can find.

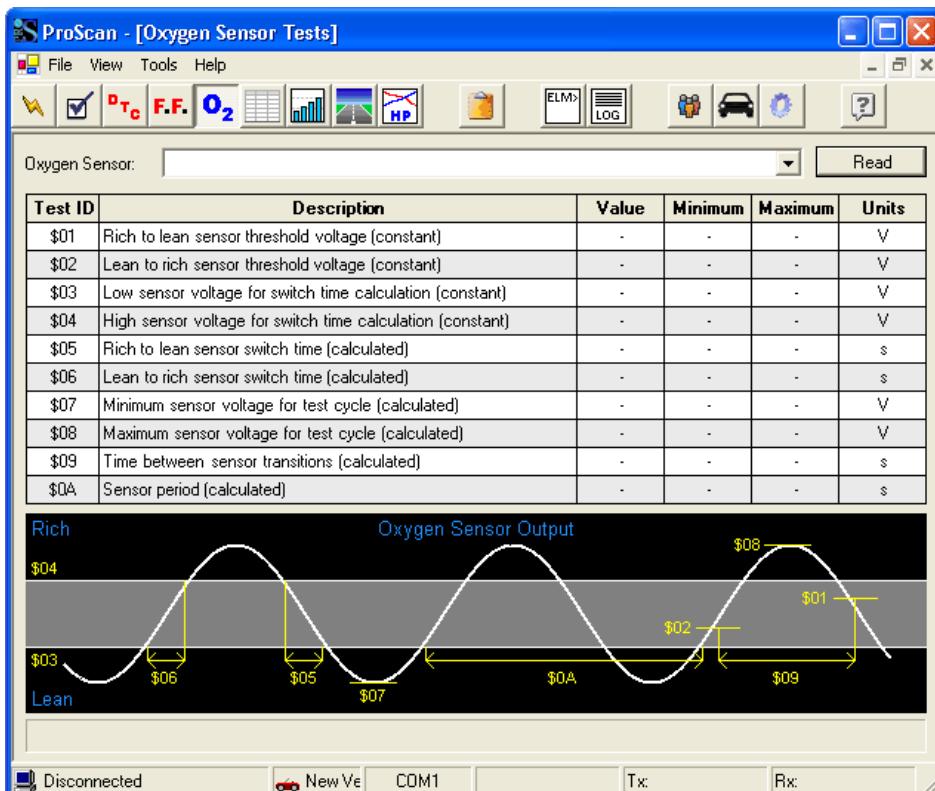
The TOAD software counts with the Fuel Economy Tool. This car diagnostic tool helps to adapt driving behaviour to reduce the fuel consumption of a vehicle. The tool could be launched during a test drives while buying a car. After a quick drive the analysis would be available and one could see the real fuel consumption data for the car based upon each driver's habits. This feature can help to make a best choice when buying a car and prevent from large costs. Moreover, there are no restrictions on how much data users can log and for how long because the Vehicle Manager can keep track of multiple vehicles and owners simultaneously. [19]

CHAPTER 3: AVAILABLE TOOLS



The Freeze Frame Data tool allows to monitor any data at any period by saving all the previous information. It also shows all detected problems and helps to resolve them. Users can use Data Logger to record healthy engine parameters and use that data to help spot problems in the future.

Figure 3.1.9a – Freeze Frame Data with TOAD



The Oxygen Sensor Tests allow to display all tests the car had for all individual oxygen sensor. If the car has a problem, this tool will help to determine if the oxygen sensors have fails and detect exactly which sensor.

Figure 3.1.9b – Oxigen test with TOAD

CHAPTER 3: AVAILABLE TOOLS

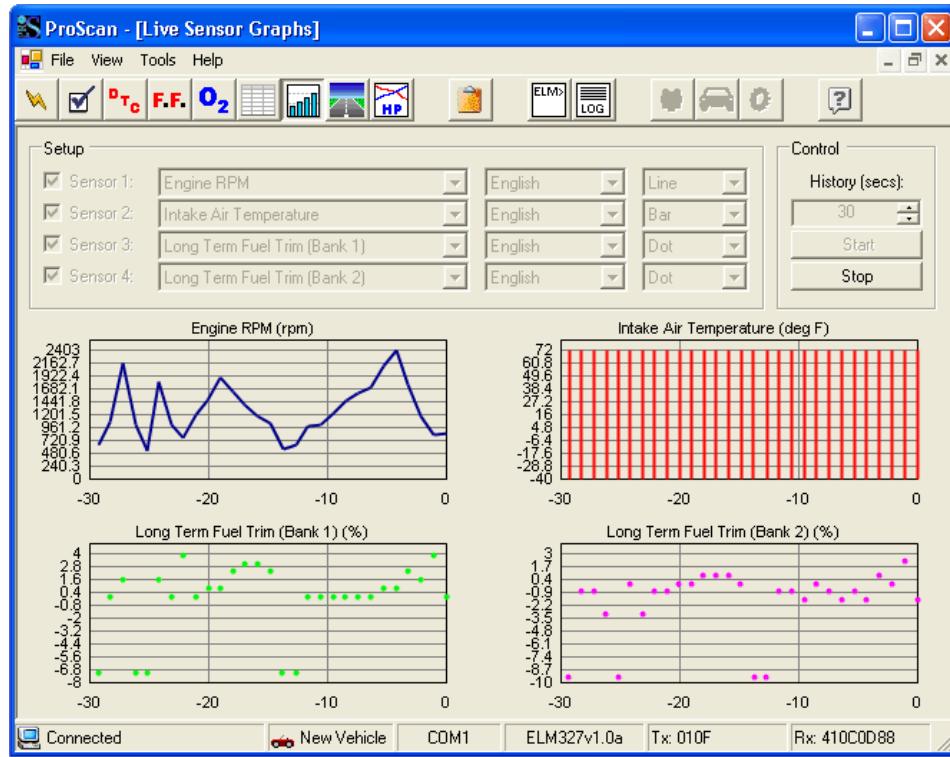


Figure 3.1.9c – Live sensor graphics in TOAD

The alerts feature produces an audible warning when a parameter value (coolant temperature, RPM, etc) goes outside normal operating range. Also, the Diagnostic Report Generator of TOAD allows to generate diagnostics report for a car with one button.

TOAD software is complete and allows to read the vehicle's diagnostic trouble codes and it also plots graphs with live vehicle sensors (including wide-band O₂ sensors). Initially it was designed to make an interface for cars easy to navigate, inexpensive and functional. In addition, it can actuate in bi-directional controls, reset adaptations, and view inspection/maintenance system test results to quickly determine what service the vehicle requires.

		OBD-II Diagnostic Report									
		<u>Prepared by:</u>				<u>Prepared for:</u>					
We analyzed your 2004 Ford Mustang Cobra on March 20, 2006 at 9:28:50 PM.											
MIL Status	Stored Diagnostic Trouble Codes										
 OFF											
Stored DTCs	Pending Diagnostic Trouble Codes										
00											
Freeze Frame Data (Not Available)											
Description	English	Units	Metric	Units	Description	English	Units	Metric	Units		
Fuel System 1 Status	-	-	-	-	Fuel System 2 Status	-	-	-	-		
Calculated Load	-	%	-	%	Engine Coolant Temp	-	°F	-	°C		
STFT - Bank 1	-	%	-	%	LTFT - Bank 1	-	%	-	%		
STFT - Bank 2	-	%	-	%	LTFT - Bank 2	-	%	-	%		
STFT - Bank 3	-	%	-	%	LTFT - Bank 3	-	%	-	%		
STFT - Bank 4	-	%	-	%	LTFT - Bank 4	-	%	-	%		
Intake Manifold Pressure	-	inHg	-	kPa	Engine RPM	-	rpm	-	rpm		
Vehicle Speed	-	mph	-	kph	Spark Advance	-	°	-	°		
Continuous & Non-Continuous Monitoring Tests											
Continuous Monitors	Supported?	Completed?		Non-Continuous Monitors		Supported?	Completed?				
Misfire	Supported	Complete		Catalyst	Supported	Complete					
Fuel System	Supported	Complete		Heated Catalyst	Unsupported	-					
Comprehensive Component	Supported	Complete		Evaporative System	Supported	Complete					
				Secondary Air System	Supported	Complete					
				A/C System Refrigerant	Unsupported	-					
				Oxygen Sensor	Supported	Complete					
				Oxygen Sensor Heater	Supported	Complete					
				EGR System	Supported	Complete					
Notes											

Figure 3.1.9d – Diagnostic report in TOAD

3.1.10 PCMSCAN

PCMSCAN is a software that can be bought for a price of \$170 directly on the website <http://www.palmerperformance.com/store>. It has connection by OBD-II port with the vehicle so in theory it should work with all car models made from year 1996 (when OBD-II port started being obligatory in vehicles). It supports all OBD-II protocols like:

- SAE-J1850 (PWM and VPW)
- ISO-9141
- ISO-14230 (KWP2000)
- ISO-15765 (CAN)

This software offers customizable log file data export to .CSV file for easy viewing in other programs like Excel. Full support for data log file bookmarks is included as well— if user notice the engine misfiring or some other problem, he can add a bookmark into the logged data that is being recorded. Then he can come back at any time and view the surrounding frames of data to analyze the problem. – [20]

PCMSCAN is able to read and clear stored Freeze Frame Data. As well, users can save and load also their log files for offline analysis.

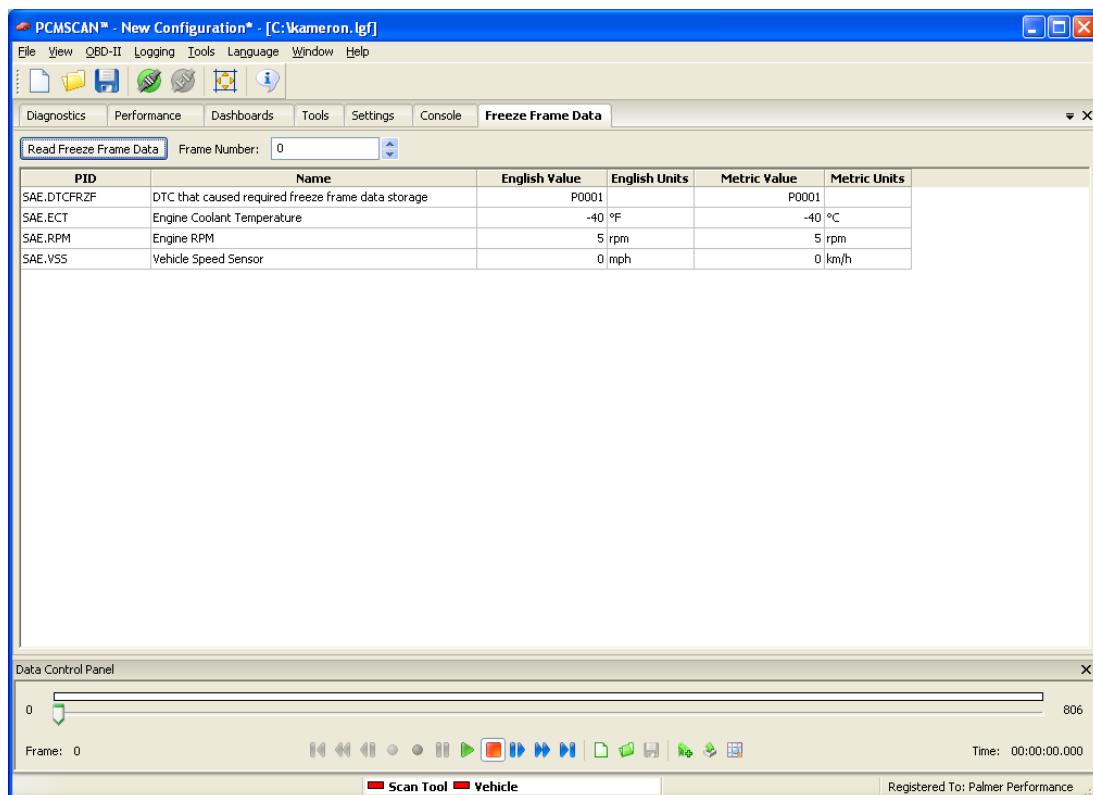


Figure 3.1.10a – Freeze Frame Data with PCMSCAN [20]

The Alert System allows users to configure the software to monitor any parameter they like. When the specified conditions are met, the software will automatically play a sound to warn customer if he is driving. It supports over 220 generic OBD-II parameters, including O2 sensors. Moreover, it can read the status of continuously and non-continuously monitored tests.

CHAPTER 3: AVAILABLE TOOLS

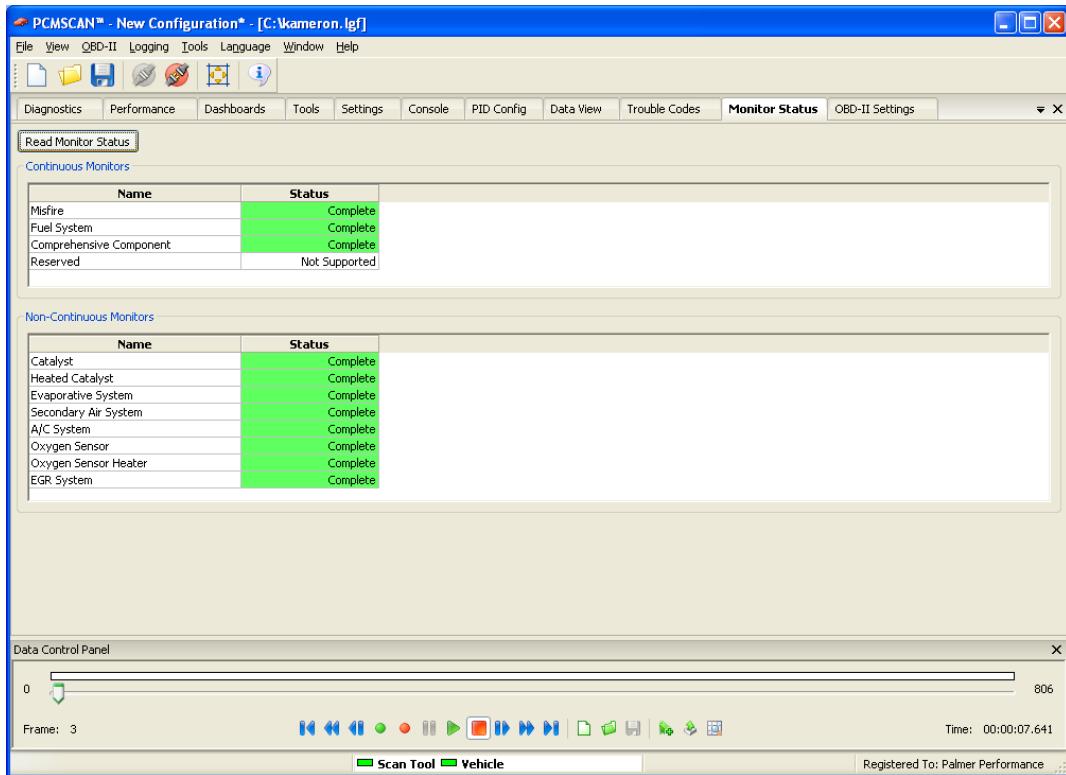


Figure 3.1.10b – Example of different tests with PCMSCAN [20]

3.1.11 CANiBUS Server

CANiBUS is a web server written in Go language by Open Garages. This server allows a room full of researchers to simultaneously work on the same vehicle, whether for instructional purposes or team reversing sessions. The Go language is portable to any operating system, but it may have issues with low-level drivers on certain platforms. For example, Go does not support the necessary socket flags to initialize the CAN interface. (This problem could be addressed by implementing socketcand, but as of this writing, that feature has yet to be implemented.) CANiBUS does have a driver for ELM327 that supports generic sniffing. [21]

3.1.12 Kayak

Kayak is an application for CAN bus diagnosis and monitoring. Kayak is a Java-based GUI for analyzing CAN traffic. It has several advanced features, such as GPS tracking and record and playback capabilities. It utilizes socketcand in order to work on other operating systems. Its main goals are a simple interface and platform independence. It is implemented in pure Java and has no platform specific dependencies. Kayak is based on SocketCAN and the connection to a SocketCAN bus takes place using TCP/IP and a socketcand server. Therefore the bus access is network transparent and can be shared with multiple users. [22]

3.1.13 Octane

Octane is an open source CAN bus sniffer and injector with a very nice interface for sending and receiving CAN packets, including an XML trigger system. Currently, it runs only on Windows.

3.2 HARDWARE

3.2.1 ELM

The ELM327 OBD2 Interface is a car diagnostic tool that is used to transmit data from OBD2 compliant vehicle to laptop computers, desktop computers, Android smartphones, Android tablets, iPhones and iPads. The technology that it provides with allows to receive real-time information from the ECU and to read and clear Trouble Codes associated with the Check Engine Light.

The car diagnostic tool can be used with most OBD2 compliant vehicles and interfaces are compatible for use with Windows XP, 7, 8, 10, OSX, iPhone, iPad and Android smartphones and tablets. Some software applications have nice graphics while others have robust logging for diagnostic purposes. These scanners can be used for professional or entertainment purposes. [23]



Figure 3.2.1 – ELM-USB connector with cable and CD with some apps

3.2.2 Delphi

When purchasing Delphi DS-150E on the website <https://www.delphiautoparts.com> a package is sent to customer. As we can see in the image below the package includes the next items:

- OBD-II adapter with light and cable to the memory
- Memory with REC button and blue/green light
- Cable USB for connecting the memory with the PC
- CD with Delphi software to be installed in a Windows computer

It is noticeable that the memory has an own battery and can work with or without being connected with the PC. If it is not connected, the battery provides the energy for saving the data that later will be sent to the PC when connecting the USB cable.



Figure 3.2.2 – Delphi DS150 memory unit, OBD-II adapter, cables and CD with software

3.2.3 Arduino

Arduino is an open-source electronics platform based on easy-to-use hardware and software. The hardware essentially consists in a board with a microcontroller. Thanks to the microcontroller, Arduino boards are able to read inputs and to turn them into outputs. At the beginning it was designed to reduce the price of the microcontrollers and to make them accessible for everyone. Since then, many projects have been developed with the help of Arduino. In contrast with some other competitors, Arduino is cheaper and both hardware and software are open-sources. Moreover, it can be used with Windows, Linux or Mac without distinction and it is easier to use for beginners so students learn how to program in C++ quickly. In the images below we can first see the Arduino main board and then mounted and connected with the OBD-II adapter.

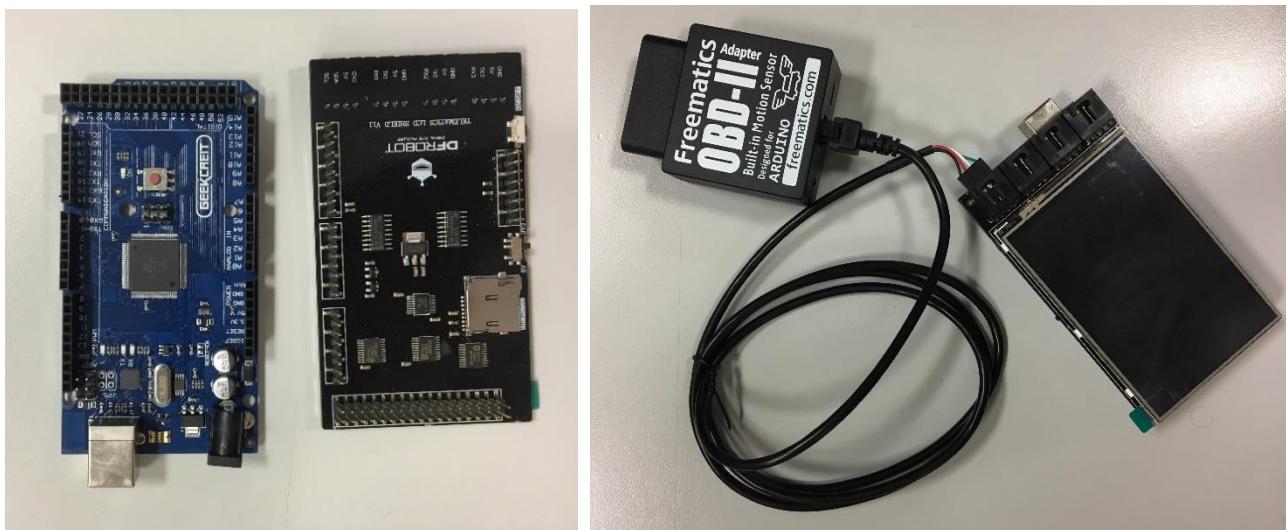


Figure 3.2.3 – Arduino board disassembled (left) and assembled with OBD-II connector (right)

3.2.4 VSCOM

VSCOM is a company that has developed many kinds of adapters for different connections. CAN adapters allow to connect a PC to CAN networks in industrial automation or automotive monitoring applications. The access is given by USB 2.0, TCP/IP or PCI Bus. By using a special library user applications control CAN networks via any of the converter models. This library supports all converters, application software for one model will directly operate with another model. The advantage is that the library supports C/C++, Delphi and LabVIEW. [24]

3.2.5 Higher-End Hardware

Higher-end devices significantly cost more money, but they are capable of receiving more simultaneous channels and offer more memory to help prevent packet loss. High-performance tools often support eight channels or more, however usually not so many channels are needed. These devices often come with their own proprietary software or a software subscription at sometimes significant added cost. If a higher-end device that specifically works with *Linux*, try *Kvaser*, *Peak*, or *EMS Wünsche*. The devices from these companies typically use the sja1000 chipset at prices starting around 400€. In any case, it is always recommended to be sure the software associated with the device we choose does what we want because we will usually be locked into their API and preferred hardware. [25]

4. EXPERIMENTS CARRIED OUT

4.1 FIRST CASE: THE AMBULANCE



In order to connect us with a vehicle with CAN for using an OBD-II scanner it is first necessary to plug it into the OBD-II port of the car. CAN is one of the transport protocols of the OBD-II specification and should be supported by most OBD-II scanners. Usually the port is located in reach of the driver, under the dashboard or hidden in the centre console. The location of the port can be easier found searching in Google an image of the specific model.

Figure 4.1a – Ambulance appearance at the beginning when we arrived



This happened to us when we wanted to connect us with the ambulance we borrowed from the hospital *Policlinico Universitario Campus Bio-Medico* in Rome. It was an old ambulance that is no longer operating so they allowed us to go there and prove what we needed.

Figure 4.1b – Looking for the OBD-II connector in the ambulance

The first problem we had was that the ambulance was too old, in fact it was an Iveco Daily 3510 from November 2002. The thing is that OBD-II connectors were not obligatory at that time in EU countries and this vehicle, in particular, did not have it. Instead we found another connector that is called *Mercedes 38 pin*, not even standardized. Even though we had a converter 38pin-OBD2 and we changed the battery of the vehicle (because it had been parked in the hospital for years) but it still did not work. After a few days and acquiring new connectors finally we realized we needed another vehicle to prove and start getting some data because the ambulance was too old in both senses: it was in poor condition and it did not have the features we needed to extract the data.

Figure 4.1c – Mercedes 38pin-OBD2 adapter



4.2 SECOND CASE: TEACHER'S PERSONAL CAR

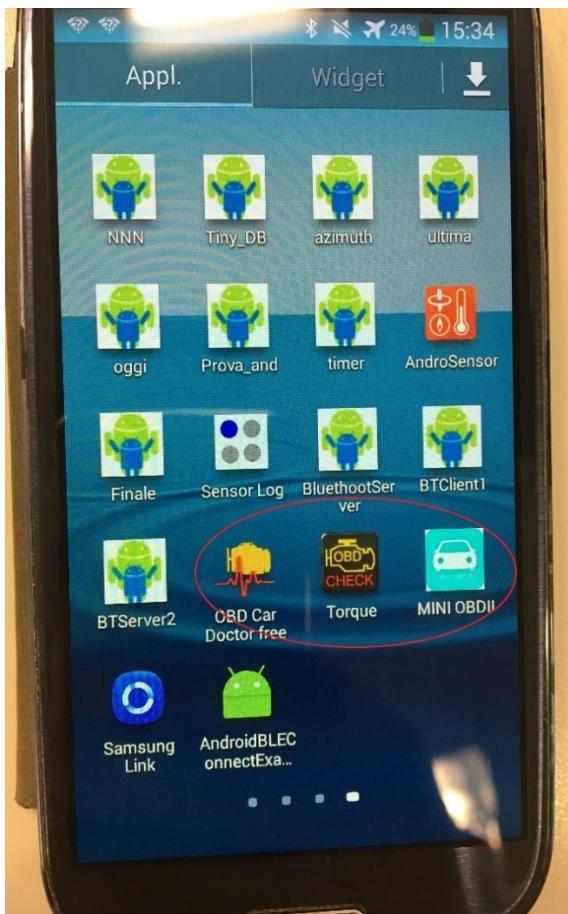


Figure 4.2 – Some apps installed in the cellular

After we spent some time looking for a new vehicle, Professor Setola offered his personal car. We could just do a quick proof but enough to know specifically what data we could get and how to do it. In this case it was very easy to access to the OBD-II connector (just under the steering wheel) so we connected to the vehicle via Bluetooth with some apps I got installed in my personal mobile phone. Particularly, these apps were *OBD Car Doctor*, *Torque* and *Mini OBDII*. With this last one we extracted many data from the car. Actually, we did not move the car but had the engine running for a few seconds and this was enough to start understanding the data we acquired. The car was a Nissan Qashqai 1.6 dCI 1300 Acenta Premium 5doors.

The proof we did with the teacher's car brought us to see specifically what kind of data we could get from the ECU (Electronic Control Unit) to start thinking about the analysis. In fact, we did not do any analysis because the data we got were not enough, but we concluded the best could be to design an algorithm that would be able to tell us something about the driving. As we had data from some variables such as revolutions/min, time driving, speed, acceleration, consume... we decided to create an algorithm that can evaluate our driving and tell us how safe it is.

4.3 PROOF WITH A RENTED CAR (FIAT 500)

On 6th June professor Setola rented a car and we started trying with all the programs we got. Firstly, we started with Delphi, and continued with other software like PCM Scan, ScanMaster-ELM, ScanTool.net, OBDTester... but the problem we got in many cases was that the version we were using was the demo version. It was like this every time: we proved several programs as I mentioned before but almost always there was something missing. In fact, we spent some money before acquiring hardware (connectors, Arduino and Delphi hardware) and software (full programs) but there was always something wrong whether in the vehicle or in our programs. **Only DashCommand was the one that we got full and was running well.**

At the beginning, Delphi would have been the most interesting because we could get data from lights sensors, air conditioned relay, etc... and initially the idea was to study some cases when a vehicle had crashed and we had to demonstrate nothing failed but the driver's attention. So, for example, if the radio and air conditioned were switched off when the accident was, it means the driver was not paying attention. Unfortunately, all features like saving data or collecting long-trip data were not available with the version we had so we had to reduce our expectations. Then we focused in an app called DashCommand because it let us connect us with the ECU via Wi-fi thanks to the ELM-327 adapter we already acquired in Amazon. But when we wanted to make proofs and get some data, the time for giving back the car was about to finish.

4.4 PROOF WITH LUCA'S CAR

Luca is one of the collaborators that Prof. Setola has in his lab. As he saw we were having many problems with the vehicles he offered me to try with his own car (Hyundai i20) and take the data from there. First of all, we started the engine and connected our adapter in the OBD2 connector. From my mobile phone I established the connection via Wi-Fi and told the app to start collecting data from the ECU while we started driving. Then we make a couple of short poofs, checked everything was working properly and we verified we could save the data.

After this, we had to reorganize a little the objectives of this thesis and we decided that the most convenient would be to go back and continue the first path we initially planned. The idea was to create some algorithms to evaluate driving. These algorithms would read the data we got (especially variables like rpm, speed, consumption...) and then give back a response with an evaluation of how safe or how efficient it was.

5. ANALYSIS

5.1 DATA WE CAN GET FROM THE VEHICLE:

Before starting directly with the analysis, let us introduce what kind of variable we have and which are the main data. There are about +100 different columns in the .csv file that was generated by DashCommand. The most important ones are below with a little explanation about.

5.1.1 Frame Number

This is just an ordinal number that indicates the position of each single measure in the table.

5.1.2 Frame Time

It is a number measured in hours, minutes and seconds (with a maximum precision of one millisecond) that indicates the exact time when the data were taken. This is a relative measure of time and starts counting from 0 when we first start the measurement. Moreover, in other columns time difference between measurements is shown in milliseconds. And when we use time we will always refer to the measures in milliseconds.

5.1.3 Forward and Lateral Acceleration

These accelerations are referred to the absolute vehicle acceleration in two directions: forward and lateral. It is measured with the mobile phone and it really corresponds to the acceleration suffered by the phone and not by the car. However, if there is no relative movement between them it has to be the same. According to the International Units System, the acceleration should be expressed in N/s^2 but here the figures are only given in Gs, which are equivalent to 9.81 N/s^2 .

5.1.4 Vehicle Roll and Pitch

This corresponds to the car's rotation around the roll and pitch axis. It is measured in degrees.

5.1.5 GPS Latitude, Longitude and Altitude

Thanks to the cellular GPS signal, it is possible to have the accurate coordinates of the position where the phone (and the car) has been moving.

5.1.6 Engine Coolant Temperature

It is the temperature of the engine coolant given both in Celsius and Fahrenheit degrees. This measure is taken directly from the ECU and is not the result of any DashCommand's algorithm.

5.1.7 Intake Manifold Absolute Pressure

As its name says, it is the absolute pressure of the intake manifold. It is measured in kPa and in inHg (mercury inches). It is also another value taken from the ECU.

5.1.8 Engine Revolutions

These are the revolutions made by the engine every instant in rpm (revolutions/min). It is value provided by the ECU.

5.1.9 Vehicle Speed Sensor

This sensor shows the instantaneous vehicle speed in mph and in km/h. This is another value provided by the ECU.

5.1.10 Intake Air Temperature

This shows the temperature (in °C but also in °F) of the intake air flow. Usually it is the same as the atmospheric temperature outside the vehicle and the value is given from the ECU without any intervention of DashCommand.

5.1.11 Air Flow Rate from Mass Air Flow Sensor

This sensor measures the quantity of air that flows into the car. It is expressed in lb/min and in g/s (grams per second).

5.1.12 Fuel Economy Calculations

This shows if the fuel consumption calculator is activated or not.

5.1.13 Trip Computer Calculations

This shows if the trip computer is activated or not.

5.1.14 CO₂ Content

It is the CO₂ gas content derived from the 'Fuel type' vehicle setting measured in lb/gal and in kg/l. It is a constant and the initial value does not change with time.

5.1.15 Fuel Density

This value is also constant in time. It refers to the density of the fuel used in the vehicle in lb/gal and g/l.

5.1.16 Stoichiometric air/fuel ratio

This is another constant that shows how much air is referred to the level of fuel. It is not measured in units but just as a division (:1).

5.1.17 Mass Air Flow

It shows how big is the mass air flow. It either uses the specified PID from vehicle settings or the best for the vehicle if no setting is provided. Figures are expressed in lb/min or in g/s.

5.1.18 Manifold Absolute Pressure

This is the pressure measured in inHg and in kPa of the manifold.

5.1.19 Fuel Flow

This is the quantity of fuel that is flowing on each instant of time. It is measured in both gallons and litres per hour.

5.1.20 Average Fuel Flow

This is the average fuel flow since last reset. It is measured in gal/h and l/h.

5.1.21 Boost Pressure Estimation

This is the boost pressure rate calculated in psi and kPa. It can also take negative values.

5.1.22 Trip Time Clock

It can measure the time since the current trip started (in hours, minutes and seconds).

5.1.23 Distance Travelled

Distance travelled in the last trip given in km and miles.

5.1.24 Current Acceleration

Current acceleration based on the last two speed readings and given in ft/s² and m/s².

5.1.25 Fuel Consumed

Total fuel consumed in the last trip in litres and gallons.

5.2 DATA ANALYSIS WITH MATLAB

As we have seen before, there are too many variables and we will have focus only in some of them. Once we finish recording the proof, the app DashCommand will generate a .csv file and we can send it to any email address. When we get the .csv file we can open it and have a look for example with Excel. Then, if we pay attention to the columns our Excel file has, we will realize there are more than a thousand of them (in fact, there are 1075 different columns). This will not allow us to work properly also because there are more than 3900 rows and, if we multiply we realize there are more than 4,000,000 different cells each one referred to a different parameter and with a different value.

Name	Value
rpm	2821x1 double
rpm1	3901x1 double
t	2821x1 double
t1	3901x1 double
time	2821x1 double
time1	3901x1 double
v	2821x1 double
v1	3901x1 double
vueltadeportiva	2827x1075 table 2820 3975 1 1

In conclusion, with so many data shown in a simple but huge table it is impossible to understand anything. Therefore, we will put the data into Matlab and then create different vectors. Like this, we will work with vectors and not with such a big table. Each column of the table will be transformed in a different vector and all values contained in will correspond to the same property. For example, we will create a vector with the numbers contained in the column *speed in km/h* and this new vector will contain all the values of speed taken in the proof.

Figure 5.2a – Vectors in MATLAB

Now, it will be much easy to organize and to work with all the data. Moreover, we will work only with the data we want to and not with all of them. Taking advantage of the previous example, it will not be necessary to create any vector from the column *speed in mph*, so we could say Matlab offers us to discard and forget all the data we do not want.

In fact, the vectors we will work with correspond to the next variables: time (sec and ms), linear speed (km/h), engine rotation speed (rpm), forward and lateral acceleration (Gs), fuel consumption (l/100km), latitude and longitude (coordinates). It is convenient to say that, as we did several proofs with Luca's car, there are several vectors for each variable (one for each proof).

Once we build these vectors we can start plotting them to see how they are. In the image below, we can a graphic that shows the engine rotation speed (in rpm) in the Y axis with the time in the X axis. These data were taken on a simple proof we did inside the parking of the university.

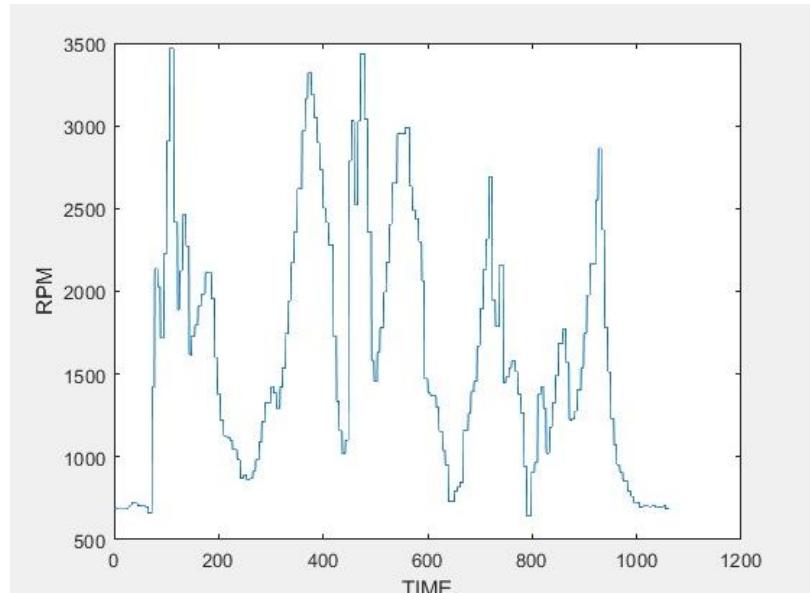


Figure 5.2b – Example of graphic in MATLAB

As we did not make the proofs always in the same place, it became necessary to show the path we had run each time. If we go further and start looking for some more complex scripts, on the Internet we can find a kind of script initially called plot_google (b). It allows us to see what way we ran and plots it in Google Maps. I personally learnt to and developed it to make the points of position change with a variable. For example, below there is an image of the way we ran in a proof and we can compare it with the ‘improvement’ I did.



Figure 5.2c – Trip path in MATLAB

Now, let us see the same but changing the points colour according to the instantaneous speed in that moment.



Figure 5.2d – Trip path in MATLAB with our improvement

Above we can clearly appreciate the improvement we did. This can be applied to all kinds of data like engine rotation speed, forward acceleration, consumption, etc. But this is especially interesting when we plot the lateral acceleration:

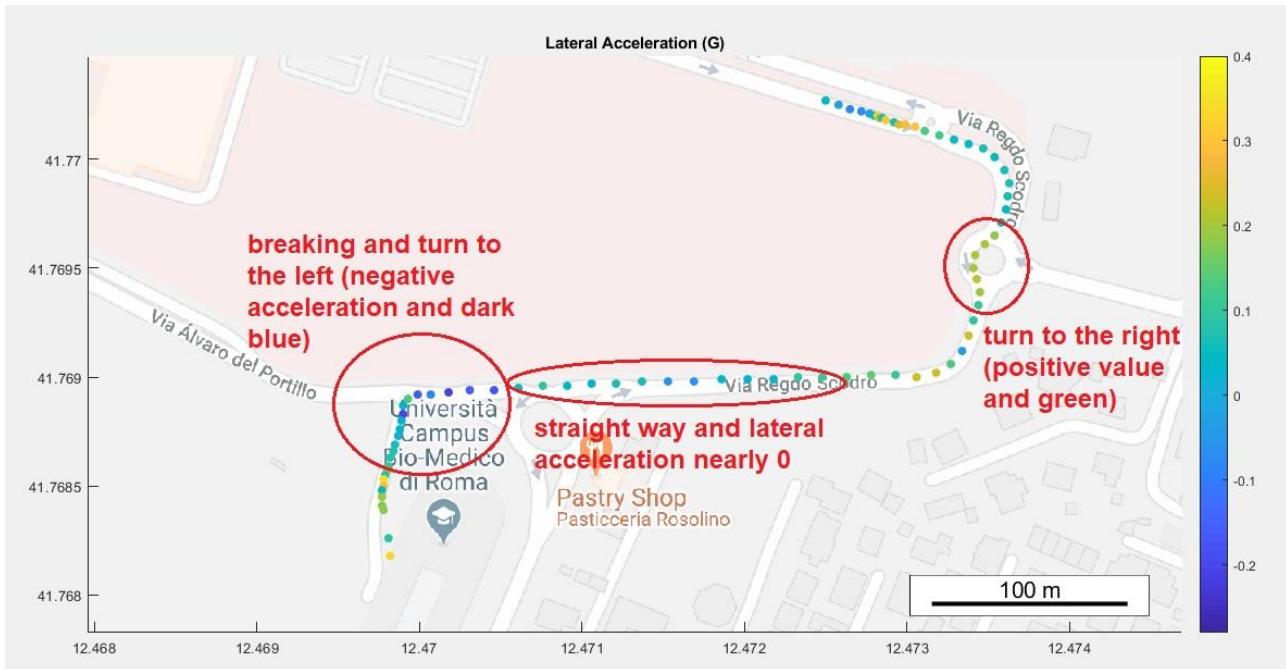


Figure 5.2e – Trip path in MATLAB with improvement and colour bar

Here, in this case we can see how the lateral acceleration changes its colour. It is very significant in the bends when it changes the value depending on the direction: if the car turns to the left it has negative values while if it turns to the right the values become positive.

5.3 ALGORITHMS DEVELOPED

As I mentioned before, we have different vectors and each of them represents a variable. The most important when developing the algorithms have been time (represented in Matlab with t), speed (v), forward acceleration (a_{for}), lateral acceleration (a_{lat}), fuel consumption ($fuel$), latitude ($lati$), longitude ($longi$) and engine rotation speed (rpm). With combination of different algorithms we can obtain some others more complex and complete. But in order to understand the reasoning let us see one of the simplest, for example the time's algorithm.

Here, as we see, we start with c/c because we are going to count if the driving time exceeds the recommendation of 2 hours. The only one vector used has been vector time, represented with t . Then, we have fixed the limit (2h in seconds are 7200s) and a counter that will change its value every time a condition is met. Moreover, it has been necessary to set the condition: ‘if the value of time vector exceeds 7200s, the value of our counter will be 1’ and, ‘if the condition is not met, counter value will be 0’. In addition, we wanted system to tell us in a sentence if our driving time has met the experts’ recommendation so we created another algorithm: ‘if the value of our counter is 1 (because it exceeds 7200s), show <>too much time driven>> and ‘if is not 1, then show <>properly time driven>>’.

CHAPTER 5: ANALYSIS

```

clc
limittime = 7200;
counter_t = 0;
for i = 2 : length(t)

    if(t(i)>= limittime && (t(i-1)<limittime) )
        counter_t=1;
    end

end
if(counter_t == 1)
    'error: too much time driven!'
else
    'properly driven time'
end

```

Figure 5.3a – Time algorithm in MATLAB

Time's case has been one of the easiest, but it is a good example to know the main working principle: *if*. Most of the algorithms have the conditions established with *if* or *if-else*. If we see another more complex algorithm, for example fuel consumption, we can quickly appreciate the answer is not a sentence but a value.

```

clc
limitfuel = 15; ← Limits are set
counter_fuel = 0;
limitfuel_inf = 10;
if(mean(fuel)>=limitfuel_inf)
for i = 2 : length(fuel)
    if(fuel(i)>= limitfuel && fuel(i-1)<limitfuel)
        counter_fuel = counter_fuel +1; ← Increase/reduce the
    end
    if(fuel(i)<= limitfuel_inf && fuel(i-1)>limitfuel_inf)
        counter_fuel = counter_fuel -1; ←
    end
    if(counter_fuel<=0)
        counter_fuel=0;
    end
end
if(mean(fuel)<=limitfuel_inf)
    counter_fuel=0;
end
counter_fuel

```

Figure 5.3b – Time algorithm in MATLAB

In this case, the structure is very similar to the previous one: it starts with *clc*, then sets a counter and two limit values. Later *for* condition is used. With *for*, it is necessary to establish the condition that it will have to analyse each single data from the beginning of the vector *fuel* till the end. It will use an identifier *i* to recognize the order of each value in the vector.

The objective of this algorithm is to count how many times fuel consumption has been over the limit's range. There are two limits, the highest limit is 15l/100km and the lowest is 10l/100km. In our model car, the inner town consumption normally rounds 12l/100km so we will consider **consumption lower than 10l/100km as a positive fact that will reduce the counter value** in one unit. The same logic will be with the highest limit because **consumption over 15l/100km will increase counter** one unit each time is exceeded.

At the end the counter value will be the result of a subtraction between the number of times the superior limit is exceeded minus the number of limits the inferior limit is passed. Therefore, a high counter's value means there is an overconsumption in our driving while a low (or negative) value means efficiency in driving. It is convenient to explain that the last conditional *if* has been added to avoid the trivial case when the counter's value is negative. It is also noticeable to explain that **the counter will change its value only in the cases when the average is over the lower limit**. When it is below, counter's value will be 0.

If we apply this reasoning to design algorithms of the other variables like speed or acceleration, we obtain different counter values that mean how safe/efficient our driving was. So once we can have a singular value for each driving proof, it is time to develop an algorithm able to combine them and get an overall driving score.

First we will start with the development of the algorithm that will measure the efficiency. We will start with three simple vectors with which we have previously made an algorithm. These are fuel consumption, forward positive acceleration and engine rotation speed. As we will see now, all these last algorithms have a counter value as an answer.

For example, if we have a look into the advance acceleration (the positive values of the forward acceleration vector) algorithm, we can appreciate in the graphic there are some peaks. Usually, big acceleration peaks are due to the driver behaviour when he accelerates and breaks too much. This provokes a higher fuel consumption and it can be also dangerous due of the breakings. Moreover, passengers can start feeling uncomfortable or get dizzy.

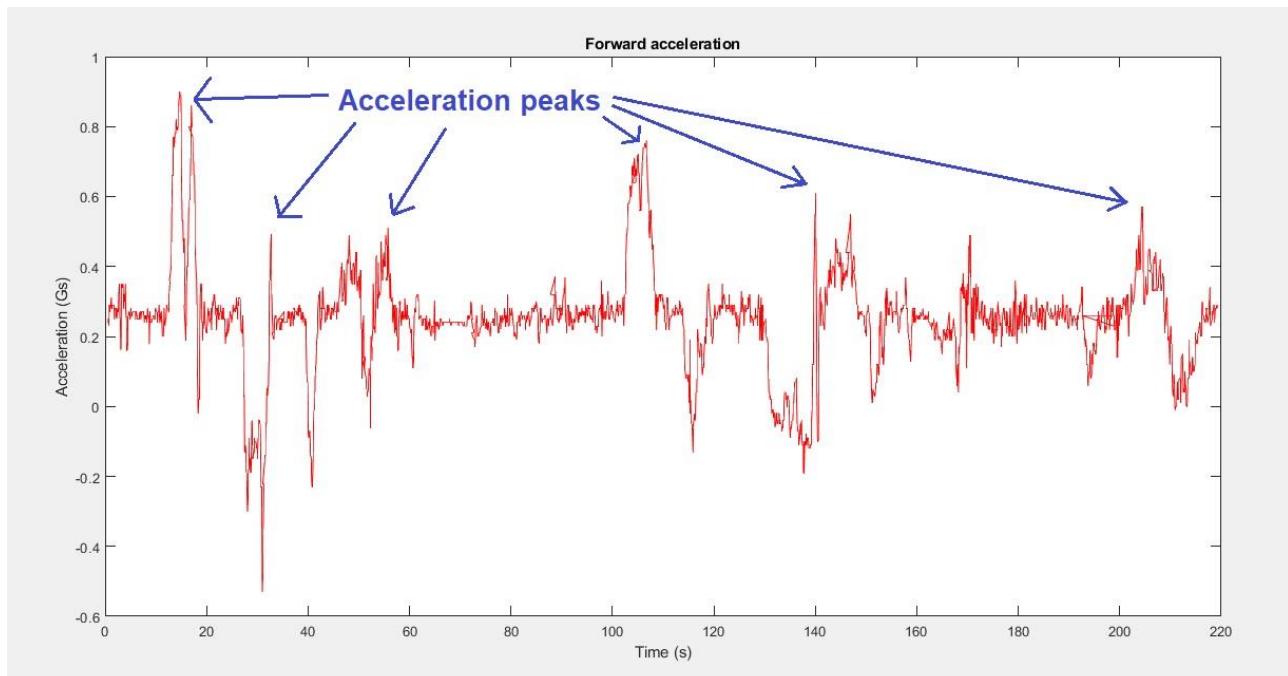


Figure 5.3c – Forward acceleration graphic

So in this case we will see how to create an algorithm able to tell us if there have been too many acceleration peaks during the driving. This algorithm is very similar to the previous one: there a counter grew each time the fuel limit was exceeded and here we will see something similar. So now the objective will be to count how many times the vehicle has suffered a big acceleration. We will establish the limit in 0.5G, which are approximately 4.8m/s^2 . Then, the counter must tell us how many times the limit has been exceeded.

```

clc
limitafor_pos = 0.5; ← Limit (0.5G)
counter_aforpos = 0;

for i = 2 : length(afor)

    if(afor(i)>= limitafor_pos && (afor(i-1)<limitafor_pos) )
        counter_aforpos = counter_aforpos +1;
    end

end
counter_aforpos ← Show counter's value

```

Figure 5.3d – Forward acceleration algorithm

Once we have learnt to create and develop algorithms, we proceed to combine some of them to create the efficiency algorithm that we mentioned before. We will start from three counters: the number acceleration peaks, the number of times engine rotation speed exceeds 2000rpm and the fuel counter we set out above. First, we will put those three algorithms all together but we will delete the last line when Matlab answers with the counter's value. Instead, we will develop some formulas to combine the variables and get a global evaluation mark. The evaluation mark will be out of 10 with the next relation: 20% based in forward acceleration, 50% for the fuel consumption and 30% for engine rotation speed. Meanwhile, the formulas, depending on the three different counter's values are:

$$\text{Forward acceleration mark (out of 2)} = 2 - 40 \times \frac{\text{value of the acceleration peaks counter}}{\text{driving time}}$$

$$\text{Fuel consumption mark (out of 5)} = 5 - 180 \times \frac{\text{value of the fuel consumption counter}}{\text{driving time}}$$

$$\text{Engine speed mark (out of 3)} = 3 - 30 \times \frac{\text{value of the engine speed counter}}{\text{driving time}}$$

If we write the formulas altogether we will obtain an equation like this:

$$\sum \text{Efficiency} = 10 - \frac{40 \times \text{acceleration}}{\text{driving time}} - \frac{180 \times \text{fuel consumption}}{\text{driving time}} - \frac{30 \times \text{engine speed}}{\text{driving time}}$$

So, in Matlab the equation is:

```
10-40*counter_aforpos/max(t)-180*counter_fuel/max(t)-30*counter_rpm/max(t)
```

Figure 5.3e – Efficiency equation

CHAPTER 5: ANALYSIS

Now, as we know all the variables and formulas for the efficiency algorithm, let us create it and show the different parts of it. With the help of the colours we can easily identify what part belongs to each algorithm:

```

clc
limitafor_pos = 0.5;
counter_aforpos = 0;

for i = 2 : length(afor)
    if(afor(i)>= limitafor_pos && (afor(i-1)<limitafor_pos) )
        counter_aforpos = counter_aforpos +1;
    end
end

```

Algorithm forward acceleration


```

limitfuel = 15;
counter_fuel = 0;
limitfuel_inf = 10;
if (mean(fuel)>=limitfuel_inf)
for i = 2 : length(fuel)
    if(fuel(i)>= limitfuel && fuel(i-1)<limitfuel)
        counter_fuel = counter_fuel +1;
    end
    if(fuel(i)<= limitfuel_inf && fuel(i-1)>limitfuel_inf)
        counter_fuel = counter_fuel -1;
    end
    if(counter_fuel<=0)
        counter_fuel=0;
    end
end
end
if (mean(fuel)<=limitfuel_inf)
    counter_fuel=0;
end

```

Algorithm fuel consumption


```

limitrpm = 2000;
counter_rpm = 0;
for i = 2 : length(rpm)
    if(rpm(i)>= limitrpm && rpm(i-1)<limitrpm)
        counter_rpm = counter_rpm +1;
    end
end

```

Algorithm engine speed (rpm)


```

if (mean(fuel)>=limitfuel) ← Average condition
efficiency_mark=5-40*counter_aforpos/max(t)-30*counter_rpm/max(t)
else
efficiency_mark=10-40*counter_aforpos/max(t)-180*counter_fuel/max(t)-30*counter_rpm/max(t)
end

```

Fuel counter's impact

Efficiency formulas

Algorithm efficiency

Figure 5.3f – Efficiency evaluation algorithm

In the last part (in blue) we can see there is an *if* condition for the case when the fuel consumption average is higher than the limit of 15l/100km. If the condition is satisfied, then the 50% of the mark equivalent to the consumption becomes 0 (it disappears in the formula). Therefore, the equation changes and the new maximum mark is 5 (out of 10).

Now, we will start to develop another algorithm that will work very similar to the last one. This new algorithm will have to be able to **measure the safety** of a driving. It will start with four different variables which are lateral acceleration, negative forward acceleration (breaking), linear speed and driving time.

The same as we did before, this algorithm will be composed by some other different algorithms and we will see one of them right now, for example lateral acceleration.

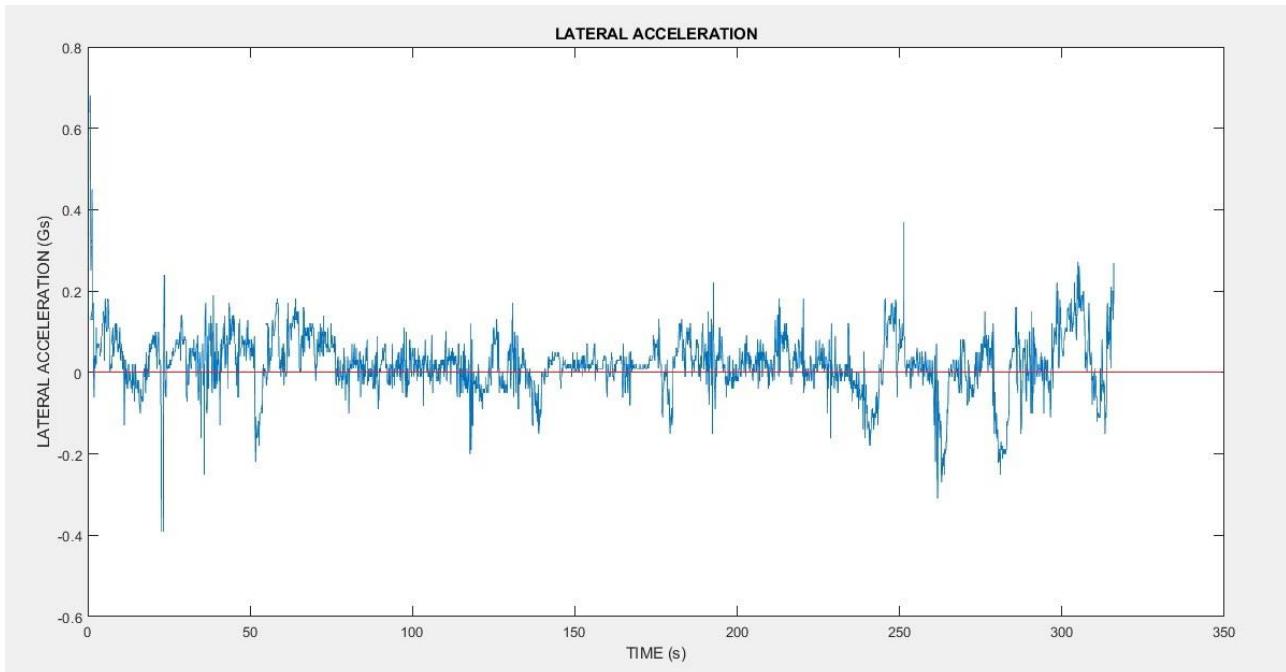


Figure 5.3g – Lateral acceleration graphic

As we can appreciate, all values are close to 0. This means there is no lateral acceleration and no turn is made. We plotted before another graphic that, with the help of the latitude and longitude coordinates, shows the lateral acceleration according to the path. Let us pay attention to the turns because depending on the direction it will have a positive or a negative value.

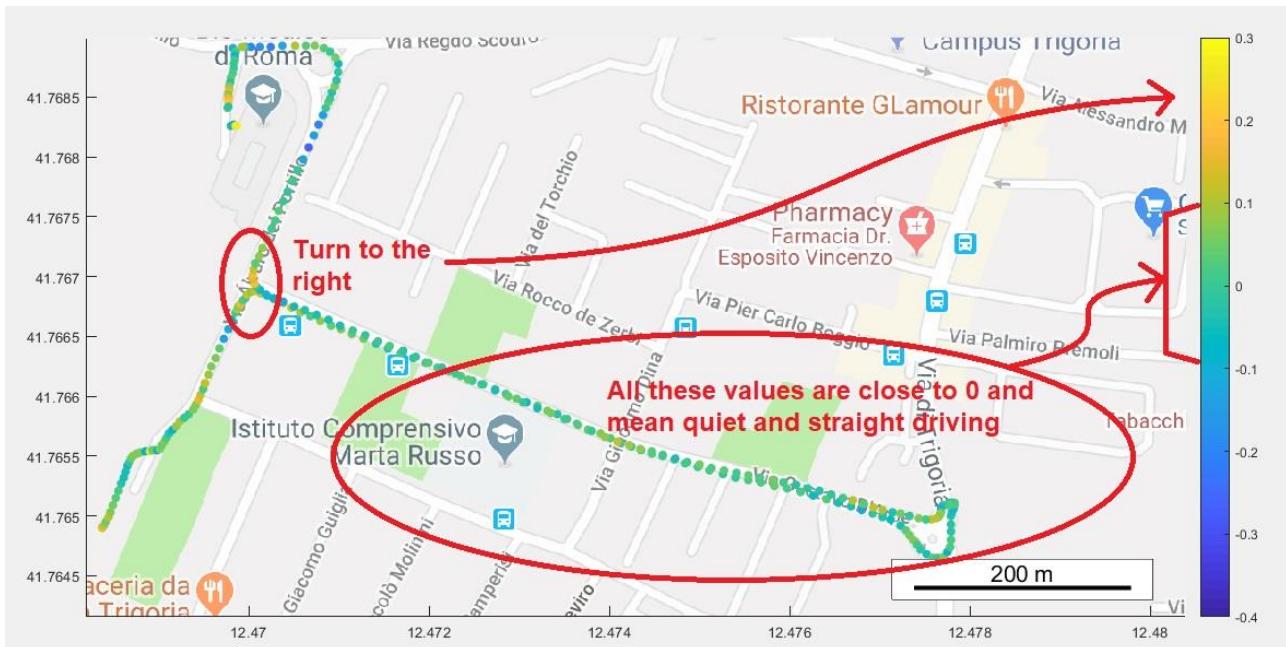


Figure 5.3h – Trip map with lateral acceleration

CHAPTER 5: ANALYSIS

In order to create the algorithm, let us establish the limit in +0.4G. With this restriction we can already elaborate it and count how many times lateral acceleration goes out the limits.

```

clc
limitalat = 0.4;
counter_alat = 0;
limitalatneg=-0.4;
for i = 2 : length(alat)

    if(alat(i)>= limitalat && (alat(i-1)<limitalat) )
        counter_alat = counter_alat +1;
    end
    if(alat(i)<= limitalatneg && (alat(i-1)>limitalatneg) )
        counter_alat = counter_alat +1;
    end

end
counter_alat

```

Figure 5.3i – Lateral acceleration algorithm

It is noticeable that we have to build two different algorithms with the same variable: forward acceleration. From now we will distinguish between negative forward acceleration and dangerous acceleration. Actually, both of them are negative values of the forward acceleration vector. The different is only in the limit of their respective algorithms.

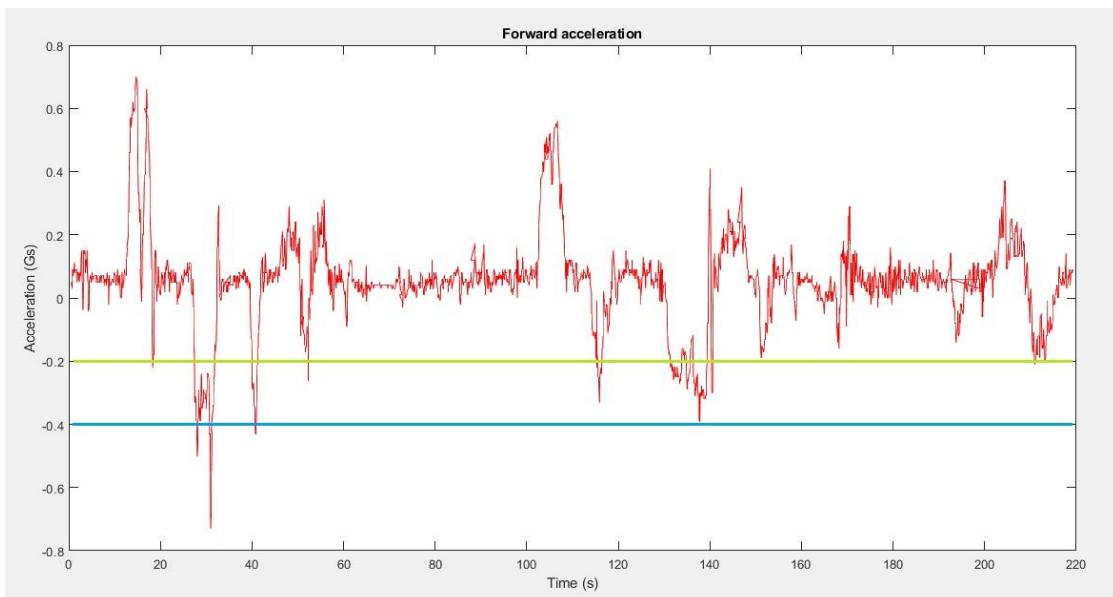


Figure 5.3j – Forward acceleration graphic

Negative forward acceleration's algorithm has a limit of -0.2G. In the case of the image above we can see many values are under the green line. Nevertheless, there are peaks of more than 0.6G negative acceleration which cannot be considered the same as the previous ones because these last ones are much more dangerous. For this reason, we have to create another algorithm to count the points when the dangerous

limit of -0.4G is exceeded (under the blue line). Below we can see the algorithm.

```

clc
limitdangafor = -0.4;
counter_dang_afor = 0;

for i = 2 : length(afor)

    if(afor(i)<= limitdangafor && (afor(i-1)>limitdangafor) )
        counter_dang_afor = counter_dang_afor +1;
    end

end
counter_dang_afor

```

Figure 5.3k – Dangerous acceleration algorithm

CHAPTER 5: ANALYSIS

If we continue with the other variables we end up in the main safety algorithm. For this, it is necessary to establish the percentages of each variable in order to evaluate the driving. The percentages are 20% for lateral acceleration, 30% for dangerous acceleration, 20% for negative forward acceleration, 30% for linear speed and 10% extra for time. Below, the formulas are shown depending on the five different counter's values:

$$\text{Lateral acceleration mark (out of 2)} = 2 - 30 \times \frac{\text{value of the lateral acceleration counter}}{\text{driving time}}$$

$$\text{Dangerous acceleration mark (out of 3)} = 3 - 200 \times \frac{\text{value of the dangerous acceleration}}{\text{driving time}}$$

$$\text{Negative acceleration mark (out of 2)} = 2 - 50 \times \frac{\text{value of the negative acceleration}}{\text{driving time}}$$

$$\text{Linear speed mark (out of 3)} = 3 - 80 \times \frac{\text{value of the linear speed counter}}{\text{driving time}}$$

If we write the formulas altogether we will obtain an equation like this:

$$\sum \text{Security} = 10 - \frac{30 \times \text{lateral acc.}}{\text{driving time}} - \frac{200 \times \text{dangerous acc.}}{\text{driving time}} - \frac{50 \times \text{negative acc.}}{\text{driving time}} - \frac{80 \times \text{speed}}{\text{driving time}}$$

The equation, written in Matlab language, would be:

```
10-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_afornegl/max(t)-30*counter_alat/max(t)
```

Figure 5.3l –Safety equation

Another particularity of this algorithm is time. From the beginning we will not consider it as an important safety value. But what experts say is that driving more than two hours without any stop becomes affecting driver's reflexes and attention. For this reason, and although our algorithm has been designed to evaluate only inner town driving, we will give a **10% extra considering time effects**. The formula, in case time exceeds 2h will be:

```
9-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_afornegl/max(t)-30*counter_alat/max(t)
```

Figure 5.3m – Safety equation considering time excess

The effect of time will be easier to implement because it will remain neutral or, if the 2h limit is exceeded it will subtract one unit to the evaluation mark. To add the time consideration to the main algorithm we will have to use conditional *if* as you can appreciate below.

```
if counter_t==0
10-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_afornegl/max(t)-30*counter_alat/max(t)
else
9-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_afornegl/max(t)-30*counter_alat/max(t)
end
```

Figure 5.3n – Safety equation algorithm depending on time

Now, in the next image we can see the safety algorithm. This has been composed with different algorithms (or part of them) developed to consider all the variables.

```

cic
limitalat = 0.4;
counter_alat = 0;
limitalatneg=-0.4;
for i = 2 : length(alat)
    if(alat(i)>= limitalat && (alat(i-1)<limitalat) )
        counter_alat = counter_alat +1;
    end
    if(alat(i)<= limitalatneg && (alat(i-1)>limitalatneg) )
        counter_alat = counter_alat +1;
    end
end

```

Safety algorithm


```

limitdangafor = -0.4;
counter_dang_afor = 0;
for i = 2 : length(afor)
    if(afor(i)<= limitdangafor && (afor(i-1)>limitdangafor) )
        counter_dang_afor = counter_dang_afor +1;
    end
end

```

Lateral acceleration algorithm


```

limitafor_neg=-0.3;
counter_aforneg = 0;
for i = 2 : length(afor)
    if(afor(i)<= limitafor_neg && (afor(i-1)>limitafor_neg) )
        counter_aforneg = counter_aforneg +1;
    end
end

```

Dangerous acceleration algorithm


```

limittime = 7200;
counter_t = 0;
for i = 2 : length(t)
    if(t(i)>= limittime && (t(i-1)<limittime) )
        counter_t=1;
    end
end

```

Time algorithm


```

limityv = 51;
counter_v = 0;
for i = 2 : length(v)
    if(v(i)>= limityv && (v(i-1)<limityv) )
        counter_v = counter_v +1;
    end
end

```

Linear speed algorithm (m/s)


```

if counter_t==0
10-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_aforneg1/max(t)-30*counter_alat/max(t)
else
9-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_aforneg1/max(t)-30*counter_alat/max(t)
end

```

Formulas with time condition included

Figure 5.3o – Safety evaluation algorithm

Finally, as we have two different algorithms able to measure efficiency and safety it is very interesting if we can combine both marks and get an overall mark. The easiest is to get it as an average where each one represents 50% of the final mark. So let us see how the formula is:

`driving_mark=(security_mark+efficiency_mark)/2`

Figure 5.3p – Global evaluation formula

CHAPTER 5: ANALYSIS

Now, depending on the mark value, we will give a written evaluation (good/bad) and an advice for the user.

```

if driving_mark<=2
'Very bad driving. Driver needs to improve immediately'
end
if driving_mark<=4 && driving_mark>2
'Bad driving. Driver must improve.'
end
if driving_mark<=6 && driving_mark>4
'Regular driving, however it could be improved.'
end
if driving_mark<=8 && driving_mark>6
'Good driving, although it could be slightly improved.'
end
if driving_mark>8
'Excellent driving'
end

```

Figure 5.3q – User's advice algorithm

If we compile both main algorithms in order to get the global evaluation's one, it will be like this:

```

clc
limitalat = 0.4;
counter_alat = 0;
limitalatneg=-0.4;
for i = 2 : length(alat)

    if(alat(i)>= limitalat && (alat(i-1)<limitalat) )
        counter_alat = counter_alat +1;
    end
    if(alat(i)<= limitalatneg && (alat(i-1)>limitalatneg) )
        counter_alat = counter_alat +1;
    end
end

limitdangafor = -0.4;
counter_dang_afor = 0;
for i = 2 : length(afor)
    if(afor(i)<= limitdangafor && (afor(i-1)>limitdangafor) )
        counter_dang_afor = counter_dang_afor +1;
    end
end

limitafor_neg=-0.2;
counter_aforneg = 0;
for i = 2 : length(afor)
    if(afor(i)<= limitafor_neg && (afor(i-1)>limitafor_neg) )
        counter_aforneg = counter_aforneg +1;
    end
end

limittime = 7200;
counter_t = 0;
for i = 2 : length(t)
    if(t(i)>= limittime && (t(i-1)<limittime) )
        counter_t=1;
    end
end

limitv = 51;
counter_v = 0;
for i = 2 : length(v)
    if(v(i)>= limitv && (v(i-1)<limitv) )
        counter_v = counter_v +1;
    end
end

if counter_t==0
security_mark=10-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_aforneg/max(t)-30*counter_alat/max(t);
else
security_mark=9-200*counter_dang_afor/max(t)-80*counter_v/max(t)-50*counter_aforneg/max(t)-30*counter_alat/max(t);
end

limitfuel = 15;
counter_fuel = 0;
limitfuel inf = 10;

```

Safety algorithm

Efficiency algorithm

Global algorithm

Figure 5.3r – Global evaluation algorithm (I)

```

if (mean(fuel)>=limitfuel_inf)
for i = 2 : length(fuel)
    if(fuel(i)>= limitfuel && fuel(i-1)<limitfuel)
        counter_fuel = counter_fuel +1;
    end
    if(fuel(i)<= limitfuel_inf && fuel(i-1)>limitfuel_inf)
        counter_fuel = counter_fuel -1;
    end
    if(counter_fuel<=0)
        counter_fuel=0;
    end
end
if (mean(fuel)<=limitfuel_inf)
    counter_fuel=0;
end

limitafor_pos = 0.5;
counter_aforpos = 0;
for i = 2 : length(afor)
    if(afor(i)>= limitafor_pos && (afor(i-1)<limitafor_pos) )
        counter_aforpos = counter_aforpos +1;
    end
end

limitrpm = 2000;
counter_rpm = 0;
for i = 2 : length(rpm)
    if(rpm(i)>= limitrpm && rpm(i-1)<limitrpm)
        counter_rpm = counter_rpm +1;
    end
end
if (mean(fuel)>=limitfuel)
efficiency_mark=5-40*counter_aforpos/max(t)-30*counter_rpm/max(t)
else
efficiency_mark=10-40*counter_aforpos/max(t)-180*counter_fuel/max(t)-30*counter_rpm/max(t)
end

driving_mark=(security_mark+efficiency_mark)/2
if driving_mark<=2
'Very bad driving. Driver needs to improve immediately'
end
if driving_mark<=4 && driving_mark>2
'Bad driving. Driver must improve.'
end
if driving_mark<=6 && driving_mark>4
'Regular driving, however it could be improved.'
end
if driving_mark<=8 && driving_mark>6
'Good driving, although it could be slightly improved.'
end
if driving_mark>8
'Excellent driving!'
end

```

Efficiency algorithm

Average mark formula

Written evaluation and advice for driver

Global algorithm

Figure 5.3s – Global evaluation algorithm (II)

5.4 RESULTS

Once we finish the algorithms we can start proving with our data. As we had Luca's car, we did several proofs with it and we could get data from several driving. Especially we did two trips in order to try the algorithms and see how different the marks are. First, we went out with the car and drove a few minutes quietly and avoiding sharp breakings. However, the next day we tried some sporty driving with breakings, acceleration boosts and increasing the speed. So let us see how are the graphics of these two trips according to each variable.

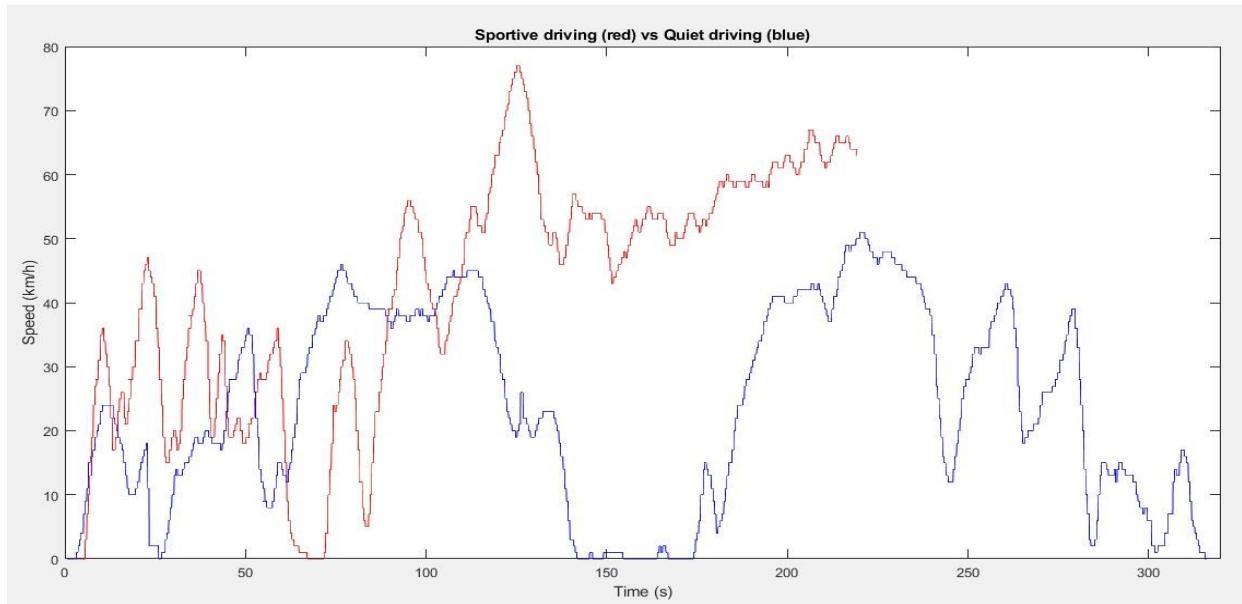


Figure 5.4a – Speed graphic

In the image of above we can appreciate how speed changes on time in the two different trips. Its noticeable we drove much faster in the sporty proof than in the quieter. Below we can see the engine rotation speed graphic.

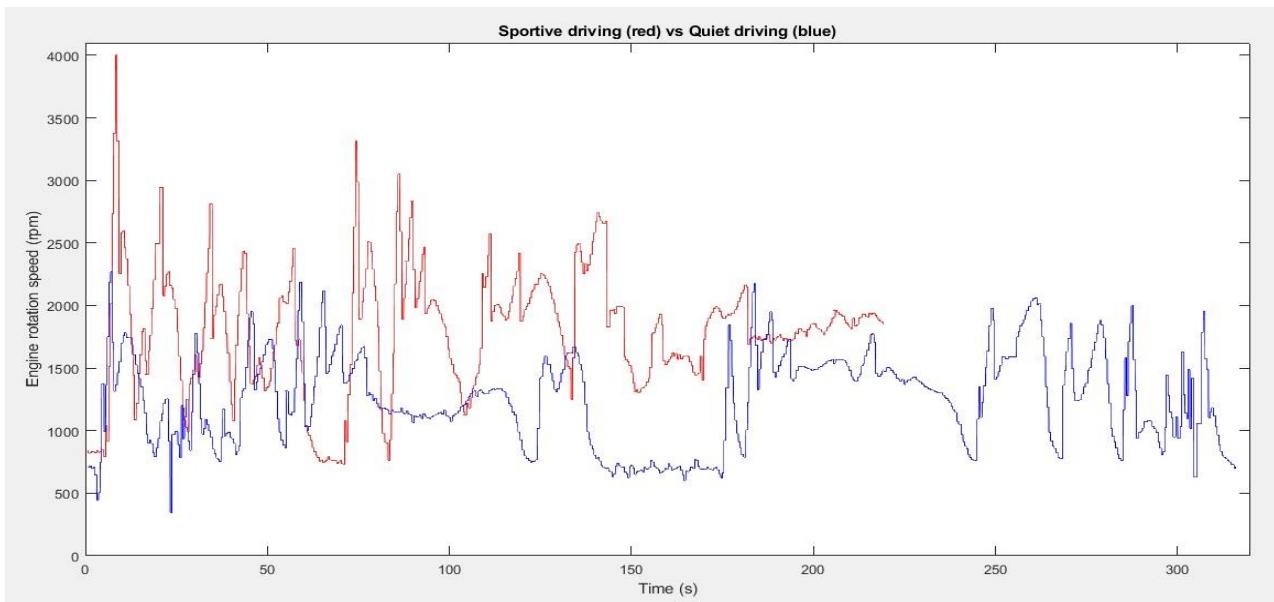


Figure 5.4b – Engine rotation speed graphic

CHAPTER 5: ANALYSIS

Here, we can easily realize there are many peaks in the red path. Peaks arrive until 4000rpm and are followed by short moments when the engine rotation speed is less than 1000rpm. This is due to the repetitive gear changes. Therefore, we cannot consider this as a normal driving because in a Diesel car the normal speed behaviour rounds 1500rpm.

The next image is the graphic of fuel consumption. It shows the instantaneous consumption of each trip. The red line is almost always over the blue line. This means the fuel consumption has been higher in the sporty trip than in the quiet one, as we can see below.

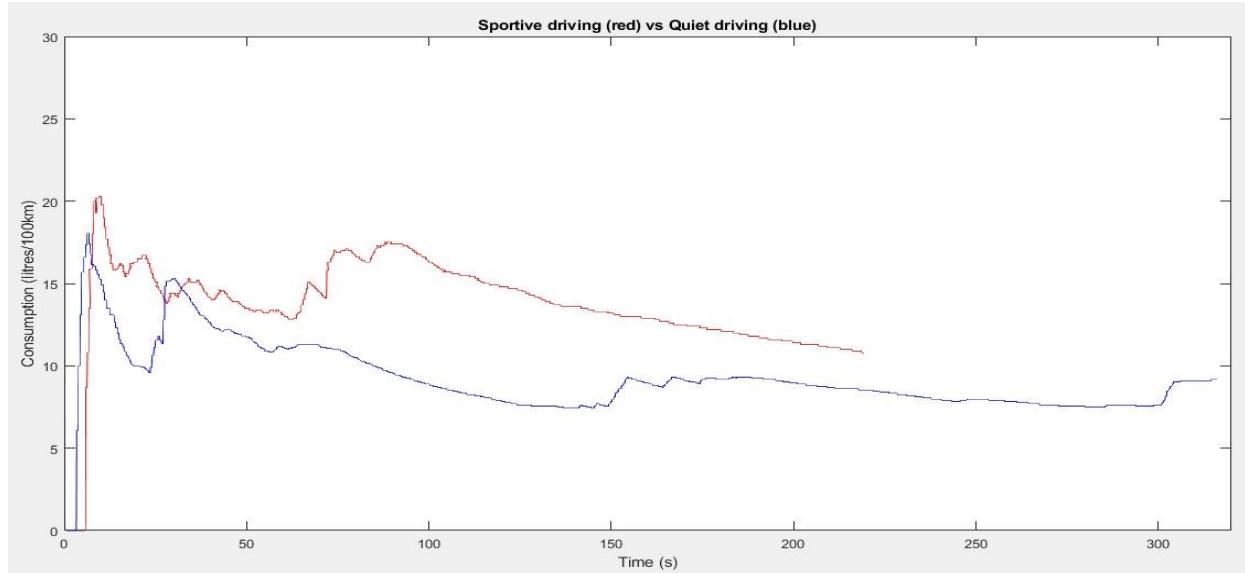


Figure 5.4c – Fuel consumption graphic

The last image corresponds to the lateral acceleration suffered by the vehicle. In the picture we can appreciate also the limits established in the algorithm. As we can check later, the instantaneous lateral acceleration (in blue) of the quiet driving never goes out these limits so the counter value will be 0 for this case. Meanwhile, if we pay attention to the sporty driving (red path) we can see it goes beyond the limits several times so the counter for this case will not value 0.

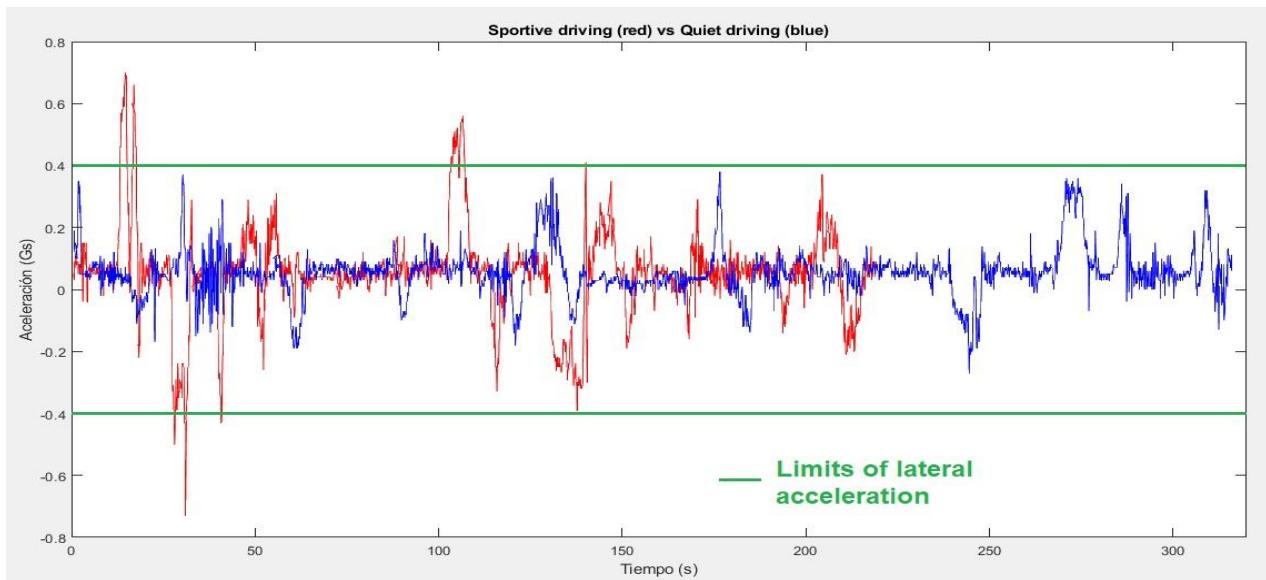


Figure 5.4d – Acceleration graphic

If we take into account the algorithms designed, in these two proofs we have strictly different marks. This is due to the different values the counters got. Now, let us see the algorithm answers.

Quiet driving proof

```
safety_mark =
9.0821

efficiency_mark =
9.5252

driving_mark =
9.3037

ans =
'Excellent driving'
```

Sporty driving proof

```
safety_mark =
0.5131

efficiency_mark =
3.3409

driving_mark =
1.9270

ans =
'Very bad driving. Driver needs to improve immediately'
```

Figure 5.4e – Quiet and sportive trip results

To see all results clearly, we collect the results of both proofs in the table below where the last rows show how the marks are when we set them into the algorithms.

VARIABLE (value of its counter)	QUIET DRIVING	SPORTY DRIVING
<i>Lateral Acceleration (out of ± 0.4 range)</i>	2	9
<i>Dangerous Forward Acceleration ($<-0.4G$)</i>	0	3
<i>Breaking Acceleration ($<-0.2G$)</i>	3	13
<i>Time (>2h)</i>	0 (OK)	0 (OK)
<i>Linear Speed (>50km/h)</i>	1	7
<i>Fuel Consumption (counter value)</i>	0	4
<i>Forward Acceleration (boosts over 0.5G)</i>	0	5
<i>Engine rotation speed (>2000rpm)</i>	5	18
<i>Driving Time (in sec)</i>	316	219
EFFICIENCY MARK (out of 10)	9.52	3.34
SAFETY MARK (out of 10)	9.08	0.51
GLOBAL MARK (out of 10)	9.3 (Excellent driving)	1.93 (Very bad driving)

Table 5.4 – Comparison of values between quiet and sportive driving tests

6. CONCLUSIONS AND FUTURE WORK

In conclusion, we can state it is possible to access and easily measure the different variables that the ECU works with. These data can be logged thanks to some software (usually apps) and can be studied later with the help of a program like Matlab. Then, the variables can be also plotted in graphics, examined in search of failures or for extra study. Moreover, if the user knows basic programming it is not so difficult to create some algorithms for the evaluation of driving. Nevertheless, the fact of programming usually needs the professor tuition. This is because sometimes there are ideas that are difficult to implement or simply the program used does not admit certain commands. Therefore, if overthinking and desperation wants to be avoided it is always better to count with the help of an expert.

If we look to the future, we realize this kind of work has already begun to develop itself. That is the case of the apps like *Drivies* or *Carmetry* that we mentioned initially. Just as an idea, if we would have liked to continue with our project, we could have developed an app able to test users driving and give back a review. Depending on the review, app users could get discounts on their car's insurance or simply follow some tips given by the app to save fuel or drive safer.

7. BIBLIOGRAPHY

- [1]<https://mechanics.stackexchange.com/questions/25561/difference-between-obdii-and-can?noredirect=1&lq=1>
- [2] <https://www.testandmeasurementtips.com/exploring-canbus-oscilloscope/>
- [3]<https://electronics.stackexchange.com/questions/121329/whats-the-maximum-can-bus-frame-message-rate-at-125-kbit-s>
- [4] pages 24 & 25 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [5] pages 27 & 28 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [6] page 20 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [7] page 33 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [8] page 23 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [9] page 24 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [10] <https://www.delphiautoparts.com/gbr/en/product/ds-series-carlcv>
- [11] <https://www.palmerperformance.com/products/dashcommand/windows/index.php>
- [12] <https://www.macupdate.com/app/mac/58857/eobd-facile>
- [13] <https://www.obdautodoctor.com/tour>
- [14] <https://www.yhasi.com/products/features.php>
- [15] <https://www.autoenginuity.com/products/scantool/>
- [16] <https://www.intrepidcs.com/products/software/vehicle-spy/>
- [17] <http://www.vetes.com.tr/intrepid-control-systems/vehicle-spy/>
- [18] <http://www.obdtester.com/pyobd>
- [19] <http://www.totalcardiagnostics.com/learn/7-best-obd2-software-reviewed-tested/>
- [20] <https://www.palmerperformance.com/products/pcmscan/index.php>
- [21] page 248 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1
- [22] <http://kayak.2codeornot2code.org/>
- [23] <https://interfusellc.com/elm327/>

CHAPTER 7: BIBLIOGRAPHY

[24] <http://www.vscom.de/usb-to-can.htm>

[25] page 245 of *Car Hacker's Handbook: A Guide for the Penetration Tester* by Craig Smith - March 2016 - ISBN-13: 978-1-59327-703-1