POLITECNICO DI TORINO

Master's Degree in Automotive Engineering

Master's Degree Thesis

Low-Cost Open-Source Data Acquisition for High-Speed Cylinder Pressure Measurement with Arduino

Supervisors

Prof. Ezio SPESSA

Prof. Dan DELVESCOVO

Candidate

Eduart CELISLAMI

April 2023

Abstract

In-cylinder pressure measurement is an important tool in internal combustion (IC) engine research and development for combustion analysis, cycle performance analysis, and knock analysis in spark-ignition engines. In a typical laboratory setup, a sub crank angle resolved (typically between 0.1^o and 0.5^o) optical encoder is installed on the engine crankshaft, and a piezoelectric pressure transducer is installed in the engine cylinder. The charge signal produced by the transducer due to changes in cylinder pressure during the engine cycle is converted to voltage by a charge amplifier, and this analog voltage is read by a high-speed data acquisition (DAQ) system at each encoder trigger pulse. The high speed of engine operation and the need to collect hundreds of engine cycles for appropriate cycle-averaging requires significant processor speed and memory, making typical data acquisition systems very expensive. The objective of this work was to develop an affordable, open-source DAQ system capable of measuring in-cylinder pressure in an ICE with Arduino. Such a system could then be applied to any engine where there is space to install an encoder on the crankshaft, and could be particularly valuable for Formula SAE teams, hobbyists, and engine builders. To this end, an absolute crankshaft encoder was installed on an Armfield CM11- MKII engine test stand, providing an absolute reference (Z) pulse once per revolution, and an incremental (B) pulse every 0.5^o CA, enabling synchronisation of pressure measurements with the engine rotation. In-cylinder pressure was measured by a Kistler piezoelectric sparkplug pressure transducer installed in the first cylinder. The transducer signal was then amplified by a Kistler charge amplifier, and the output sent to the analog input of an Arduino DUE microcontroller, while the encoder Z and B pulse signals were connected to digital input pins. Analog to Digital (ADC) readings from the Arduino are then streamed to an external SD card, enabling storage of hundreds of engine cycles worth of data. Finally, we demonstrate the range of operation, capabilities, and limitations of the Arduino DAQ system.

Ai miei genitori, a mia sorella, alla mia famiglia,
grazie ai quali sono la persona che sono.

Table of Contents

List of Tables

List of Figures

Acronyms

ICE
Internal Combustion Engine

CA
Crank Angle

CR
Compression Ratio

ECU
Engine Control Unit

MAP
Inlet Manifold Pressure

DAQ
Data Acquisition

DAC
Digital to Analog Converter

SPI
Serial Peripheral Interface

SRAM
Static Random Access Memory

TDC
Top Dead Center

ISR

Interrupt Service Routine

ADC

Analog To Digital Converter

FIFO

First In First Out

IBDC

Inlet Bottom Dead Center

IVO

Intake Valve Opening

EVC

Èxhaust Valve Closing

IVC

Intake Valve Closing

IGN

Ignition

EVO

Exhaust Valve Opening

PE

Piezo Electric

PCB

Printable Circuit Board

Chapter 1

Introduction

The objective of this thesis is to devise a cost-effective solution for the high-speed acquisition of in-cylinder pressure data employing the Arduino DUE. This undertaking is the result of a collaborative effort between the Polytechnic of Turin, Oakland University (MI), and Stellantis.

The primary aim of this project is to establish a Data Acquisition system (DAQ) capable of continuously measuring pressure levels inside the cylinders of an Internal Combustion Engine (ICE) for a few hundred cycles, storing them for later analysis of combustion variability. The present work endeavors to develop an open-source, budget-friendly DAQ system that can measure in-cylinder pressure in an ICE utilizing the Arduino. The proposed system is designed to be installed in any engine with adequate space to accommodate an encoder on the crankshaft and would be of particular interest to Formula SAE teams, hobbyists, and engine builders. In Chapter 2 the requirements for the Data Acquisition System are set, in Chapter 3 the mechanical and electrical connections necessary to realize the pressure measurements are described, in Chapter 4 the calibration methodology for the pressure transducer and the fuel injectors are explained, in Chapter 5 the Arduino DUE Code is written to allow easy distribution, in Chapter 6 the main functions used in the Matlab code are explained, in Chapter 7 the results of the tests are presented, in Chapter 8 the Future studies are explained and finally in Chapter 9 the conclusions are discussed.

1.1 Test-bench characteristics

The Test-bench engine used for this study is the CM11-MK II Gasoline Engine manufactured by Armfield Limited. It is a self-contained integrated, multi-cylinder engine, dynamometer and instrumentation system. The dynamometer consists of a brake mounted on the rear side of the engine via a flexible coupling, this

arrangement allows for small misalignments between the two shafts and isolates the brake from the vibration of the engine [1].

Figure 1.1: Armfield CM11- MK II

Source: Oakland University(MI), Energy Lab

Table 1.1: Armfield Test-bench characteristics

Manufacturer	Volkswagen
Capacity	1198 cm^3
N^o of Cylinders	3
Bore	76.50 mm
Stroke	86.90 mm
CR	10.3:1
Max Power	40kW @ 4700 rpm
Brake	Eddy current dynamometer
Max Brake power	55kW

Source: Instruction Manual CM11-MK II

In Table 1.1 the engine characteristics are shown, it is important to know that this is a Naturally Aspirated engine with a pressure range that goes from below atmospheric pressure up to 100 bar. The Test-bench is controlled remotely with the Armfield Software which allows to run the ICE in different conditions changing engine speed, engine throttle, and brake load. Engine torque is measured directly, using a strain-gauge type load cell which is connected to the brake by a load arm.

In this configuration, a Kistler Pressure Transducer is installed in the spark-plug, on the first cylinder which allows to collect the in-cylinder pressure. The system as it is allows to collect the pressure readings in function of time but not in CA degrees. This Test-bench is provided with an Emerald K3 ECU which allows to modify via software the Ignition Map, and the Injection Map. It is also possible to read the Injection timing and Manifold Absolute Pressure (MAP) which is used to allow a more uniform reading of the pressure measurements.

This test-bench enables the safe operation of an internal combustion engine within an enclosed setting. The exhaust pipe is directly connected to the building's exterior, minimizing the risk of indoor air pollution. The system incorporates safety switches that, when triggered, promptly shut down the test-bench. Furthermore, in the event of a disconnection with the remote connectivity, while the engine is running, the test-bench is set to stop the engine automatically.

To operate effectively within a closed environment, the system employs two cooling loops. The primary engine coolant is circulated by an internal pump to a heat exchanger, while secondary cooling is provided by clean water flow. To safeguard the engine's lifespan, the remote software is designed to prevent engine start-up if the water flow is not detected.

1.2 Data acquisition systems

Figure 1.2: MC USB-1808X DAQ

Source: MC Measurement Computing

To collect in-cylinder pressure measurements, it is possible to work with a ready-to-use DAQ System. Several Data Acquisition Systems are available at a wide range of prices according to how many inputs, outputs, and sampling rates. The cheapest solution which would work for our test conditions is the MC USB-1808X shown in Figure 1.2 [2], it simultaneous-sampling DAQ device with a sampling rate of up to 200k/S per channel. It is possible to connect up to 8 Analog inputs but in our application, we need to collect pressure data only from one pressure transducer. This means that the other 7 Analog Inputs are not being used, increasing the cost

of the device. This DAQ costs 989 USD[1] which increases the expenses in order to be able to collect pressure measurements at a high rate but the less expensive solutions are not capable of getting the pressure measurements for engine speeds up to 4000 RPM. Of course, these considerations can be done for activities that are closer to Formula SAE teams, hobbyists, and engine builders rather than research labs. Having said these considerations, it is possible to move on to present the Arduino DUE.

1.3 Arduino and Accesories

The aim of this research is to realize a low-cost DAQ using an Arduino DUE.

1.3.1 Arduino platform

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs and turn them into output. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing. Over the years Arduino has been the brain of thousands of projects, from everyday objects to complex scientific instruments.

Arduino also simplifies the process of working with microcontrollers, and interests other systems:

- **Inexpensive** Arduino boards are relatively inexpensive compared to other microcontroller platforms.
- **Cross-platform** The Arduino Software (IDE) runs on Windows, Macintosh OSX, and Linux operating systems. Most microcontroller systems are limited to Windows.
- **Simple, clear programming environment** The Arduino Software (IDE) is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well.
- **Open source and extensible software** The Arduino software is published as open source tools, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to.

[1]Price as displaced in MC Measurement on 24/03/2023

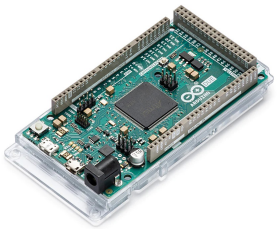
- **Open source and extensible hardware** The plans of the Arduino boards are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works and save money.[3]

After this overview, a description of the used Arduino is done.

1.3.2 Arduino DUE and accessories

The Arduino DUE is a microcontroller board based on the Amtel SAM3X8E ARM Cortex-M3 CPU, it uses a 32bit ARM core microcontroller, it has 54 digital input/output pins, 12 analog inputs each of which can provide 12 bits of resolution, 84MHz clock, and USB OTG capable connection, 2 DAC, an SPI.

Figure 1.3: Arduino DUE



Source: © 2023 Arduino

The Arduino DUE board runs at 3.3V and anything above this voltage in the I/O pin could damage the board. It has 96KB of SRAM which can be used either to store data or as a buffer. This board can be powered via USB or with an external power supply, the power source is selected automatically. It is also capable to deliver power via the 3.3V pin. It is possible to connect the Arduino DUE to the Native USB port which is connected to the SAM3X, it allows for serial (CDC) communication over USB. This provides a serial connection to the Serial Monitor or other applications on the computer [4]. In order to realize the DAQ we need some more components for the Arduino board which are written in Table 1.2.

The Adafruit Data Logger Shield is compatible with Arduino DUE, it allows for an installation of an SD card interface with FAT16 or FAT32 formatted cards, increasing the data storage capabilities of the Arduino Board. The SD Card has an 8 GB capacity, which allows storage of several pressure measurements. The Screw Terminal is used in order to connect securely the different wires coming out from the pressure transducer and the encoder, finally the container is chosen to make the system compact and easy to handle.

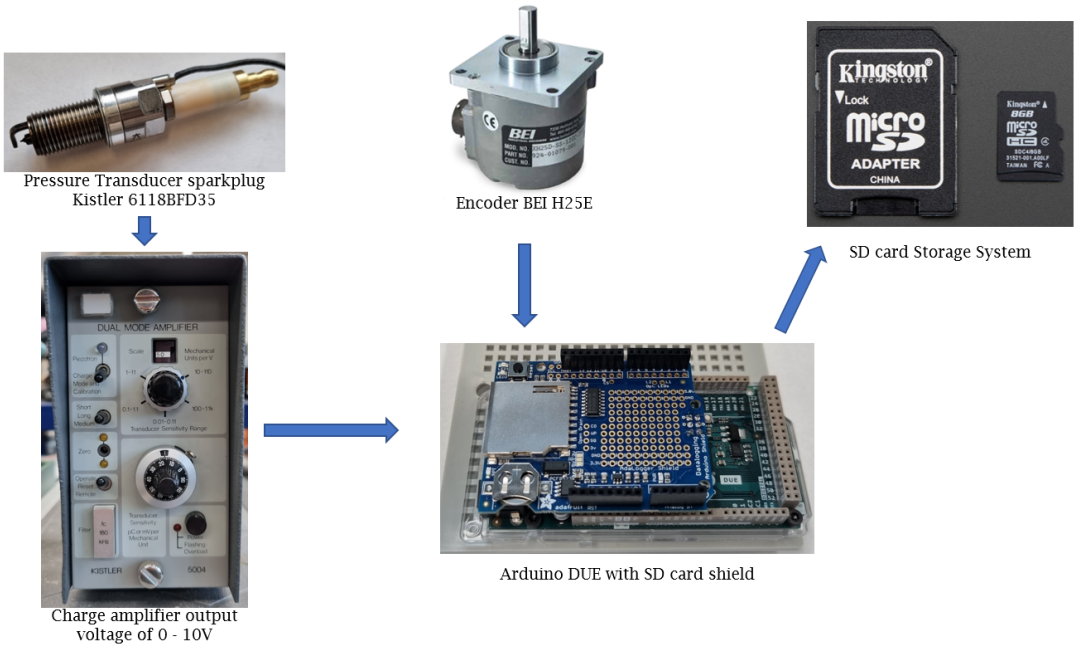
Table 1.2: Arduino DUE components and Cost[2]

Component	Cost [USD]
Arduino DUE	50
Adafruit Data Logger Shield	14
SD Card	10
SD Card Extender	10
Screw terminal	20
Container	20
Total Cost	**124**

This configuration allows powering the Encoder with 3.3V, using the quadrature signal output of the encoder to synchronize the pressure readings in function of the CA and to read the analog voltage coming from the pressure transducer.

1.4 Measurement chain

Figure 1.4: Measurement chain



The measurement chain is illustrated in Figure 1.4. To obtain in-cylinder pressure measurements, a Kistler 6118BFD35 piezoelectric transducer is installed

in the first cylinder of the Internal Combustion Engine (ICE). The pressure change inside the cylinder generates a charge in the piezoelectric pressure transducer. The Kistler 5004 Dual Mode Amplifier converts this charge into a voltage signal that ranges from 0 to 10V. This output voltage is then passed through a voltage divider to reduce the usable range from 0-10V to 0-3.3V before being fed to the Arduino.

An Encoder is required to obtain the pressure readings in relation to the engine rotation. It needs to be connected directly to the engine crankshaft to avoid errors caused by belt sliding if connected through a belt. Once both devices are installed, their output signals are connected to the Arduino DUE. The code developed later is programmed in the Arduino DUE to collect data when needed.

The collected data is then stored on the SD Card, which is installed in the Adafruit Data Logger Shield placed on top of the Arduino DUE.

Chapter 2

Requirements

After defining the required components, the performance test of the DAQ can be set based on those requirements.

2.1 Mechanical Connection

In order to properly install the encoder onto the engine crankshaft, a reliable and secure connection between the engine block and the encoder support is essential. Mounting the encoder onto the body structure of the test-bench is not feasible due to vibrations, and it is likely that running the engine in this manner would result in damage to the coupling system between the encoder shaft and the crankshaft of the engine. Therefore, it is necessary to install an encoder support directly onto the engine block, with multiple connection points to ensure stability.

Additionally, modifications to the engine pulley are required in order to establish a connection system between the encoder coupler and the engine shaft. Specifically, the center diameter must be enlarged and the bolt must be changed to accommodate the designed engine coupler.

To ensure stable mounting points for the encoder, an aluminum plate has been selected and water-cut to allow for easy installation onto the engine shaft. This mounting system is designed to allow the installation of different encoders onto the same engine.

2.2 DAQ System Requirements

In order to be able to collect the measurements that were mentioned before, some considerations have to be made on the DAQ requirements.

2.2.1 Sampling Rate

Regarding the sampling rate, the Data Acquisition System that has been designed must be capable of collecting pressure measurements starting from the idle speed and continuing up to the maximum engine speed at which the engine is intended to operate. The selection of an appropriate encoder for pressure synchronization will have an impact on the sampling rate. For instance, if a 0.5^o degree of resolution encoder is utilized, it will generate 720 pulses for every complete rotation of the crankshaft. Conversely, if an encoder with a pulse rate of 0.1^o is employed, the total number of pulses will be 3600.

Depending on the type of encoder employed, it is possible to determine the appropriate sampling rate required for the maximum available engine speed. In this particular case, the maximum engine speed is 4000 RPM. In order to ensure that no encoder increments are lost, it is necessary to have a sampling rate on the DAQ that is at least twice the sampling rate of the engine speed multiplied by the number of encoder steps per revolution as shown in 2.1.

$$\text{Sampling Rate} = 2 \left(\frac{[\text{RPM}]}{60 \frac{[\text{sec}]}{[\text{min}]}} \right) \text{N}^o \text{ of pulses} \tag{2.1}$$

2.2.2 Storage

In order to utilize the collected measurements effectively, it is important that consecutive cycles of data are acquired while the engine is operating at the various speeds. Ideally, the system should have the capacity to store a minimum of 100 continuous cycles, which corresponds to 200 engine revolutions. However, it would be preferable to store longer runs of up to 1000 cycles per experiment without any loss of data. As the data is being stored on an SD card, it is necessary to extract the card from the Arduino shield to transfer them to a computer. Hence, to minimize time losses and enhance data collection efficiency, it is desirable to store multiple experiments on a single SD card.

2.3 Pressure Transducer and Charge amplifier

On the Test-bench the Kistler 6118BFD35 Piezoelectric Pressure Transducer is installed in the first cylinder spark-plug.

Piezoelectric pressure sensors are connected to an electronic circuit which converts the charge generated by the sensor into a proportional voltage. Hence the sensitivity is given as pico-coulombs per unit of pressure (pC/bar). Pressure applied to a PE sensor produces a negative going charge signal (hence the negative sensitivity of PE sensors), which then is converted into a positive voltage signal by

the external charge amplifier. PE pressure sensors are connected to an external charge amplifier. This converts the charge into a voltage signal.

- Measurement of extremely low or very high temperatures (no electronics in the sensor)
- Adjustable measuring ranges with only one pressure sensor (measuring range adjustable in the charge amplifier)

To measure the output of the pressure transducer, the use of a charge amplifier is necessary. For this purpose, the Kistler 5004 Dual Mode Amplifier is selected as the appropriate charge amplifier for the experiment. Figure 2.1 shows the setup with the Kistler 5004 Dual Mode Amplifier already configured for the data acquisition system.

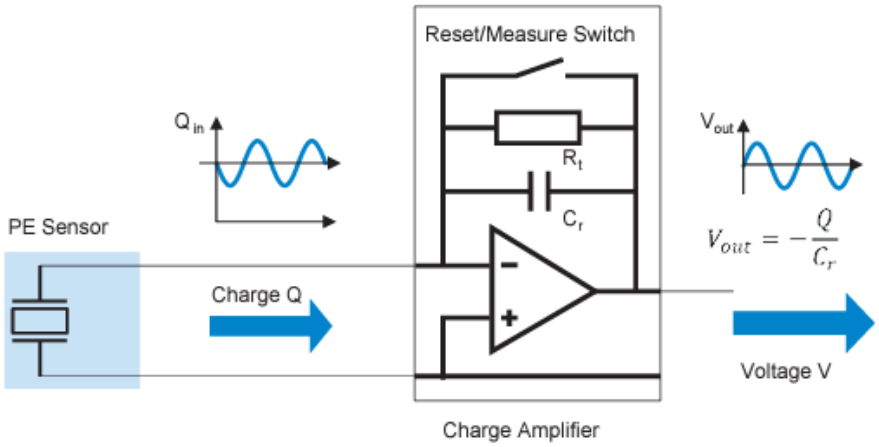
Figure 2.1: Charge amplifier Settings

Source: Oakland University(MI), Energy Lab

The amplifier converts the charge signal of the sensor into a proportional voltage signal and thus makes the measurement available for further processing [5]. The aforementioned charge amplifier provides the flexibility to adjust the sensitivity of the pressure transducer in use and optimize the output scale. This feature is particularly useful as it aids in reducing the noise in the output voltage, which is an analog signal.

In Figure 2.2 the process which converts the charge to voltage is shown, this is what the parameters are related to:

Figure 2.2: Charge amplifier Structure



Source:Kistler Catalogue

- The range capacitor Cr is used to set the measurement range of the charge amplifier. This is done by switching between different range capacitors. It is possible to measure across several decades with a high signal-to-noise ratio. Hence, for example, it is possible to use the same pressure sensor to measure pressures of a few hundred bar and a few µbar, simply by switching over the measurement range.
- The time constant resistor Rt defines the low-frequency performance of the charge amplifier. In particular, the time constant determines the cut-off frequency for the high-pass characteristic of the charge amplifier. Switching between different time-constant resistors makes it possible to change the high-pass characteristic.
- The Reset/Measure switch is used to control the start of measurement or to set the zero point.

The output voltage range of the pressure transducer goes from 0 to 10V, which means that our DAQ needs to be able to read these voltages.

Chapter 3

Design

This chapter delineates the methodology utilized to establish the system, commencing with the Mechanical Design and concluding with the Electrical Design.

3.1 Mechanical Design

This paragraph discusses the mechanical linkage between the encoder and the engine crankshaft, and the installation of the encoder support to the engine body.

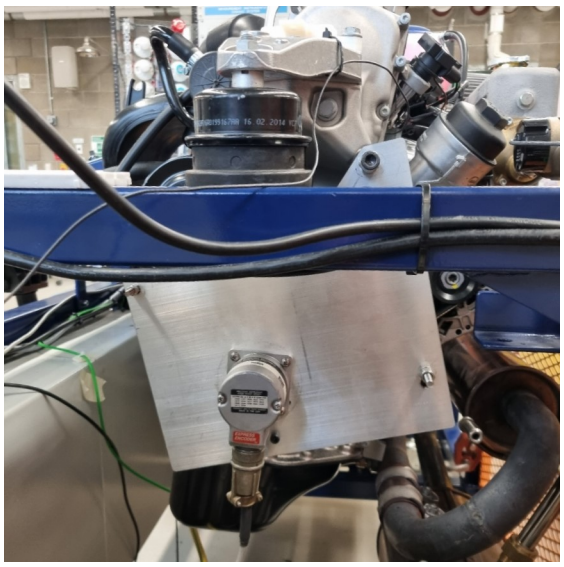
3.1.1 Crankshaft Coupling

The installation process for the encoder involves several steps, beginning with the removal of the engine pulley. This action enables the centering hole to be enlarged, facilitating the precise alignment of the crankshaft support with the engine crankshaft. By achieving concentricity between the crankshaft, the pulley, and the crankshaft support, the encoder can operate with optimal accuracy.

To accommodate the increased longitudinal length of the crankshaft supports, a longer bolt must be employed to secure the pulley. Finally, the cover is installed using the coupling system, ensuring that the encoder remains securely fastened to the engine and can perform its intended function effectively. The coupling system is a three-piece Ruland controlflex coupling. It is a lightweight, low-inertia coupling that allows speeds of up to 25,000 RPM. Controlflex couplings have a balanced design for reduced vibrations at high speeds, can accommodate all forms of misalignment, and are an excellent fit for encoder applications [6]. Furthermore, the plastic coupler reduces the impact of vibrations that emanate from the crankshaft and could affect the encoder's accuracy. The ultimate component is presented in Figure 3.2, where an increase in the axial length required to accommodate the screw fastening the pulley to the crankshaft can be observed. Mounting the encoder

on the opposite side is not a viable option due to the presence of the dynamometer that applies the load.

Figure 3.1: Rear view Encoder



Source: Oakland University(MI), Energy Lab

3.1.2 Encoder support

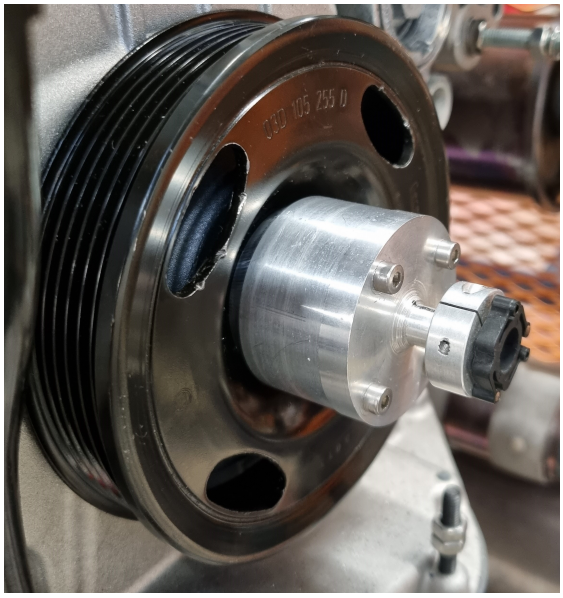
In order to properly install the encoder, it is imperative to establish a robust and unyielding connection between the encoder support plate and the engine block. This is achieved by utilizing several fastening points that prevent any movement of the encoder during engine operation. As illustrated in Figure 3.1, the plate is affixed to the engine block by four points. The plate has been fabricated with a water-jet cutter available at the Oakland University Machine Shop.

Figure 3.2 showcases the encoder's linkage to the engine crankshaft, wherein a plastic coupler is installed on the encoder shaft. This coupling permits a degree of flexibility between the two shafts, negating the necessity for perfect alignment.

Figure 3.3 provides a clear illustration of the coupling mechanism between the encoder and the engine crankshaft. The encoder shaft features a plastic coupler that allows for a certain degree of flexibility, thus obviating the need for precise alignment between the two shafts. This design decision has the added benefit of dampening vibrations that might otherwise affect the encoder's readings.

An additional precautionary measure that can be observed in Figure 3.3 is the implementation of jam nuts. This locking system serves to ensure that the encoder

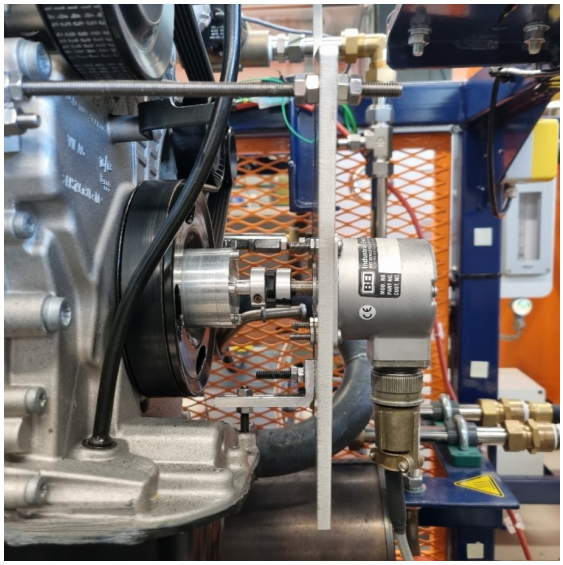
Figure 3.2: Engine Pulley



Source: Oakland University(MI), Energy Lab

support plate remains securely in position and does not become dislodged during engine operation.

Figure 3.3: Side view Encoder



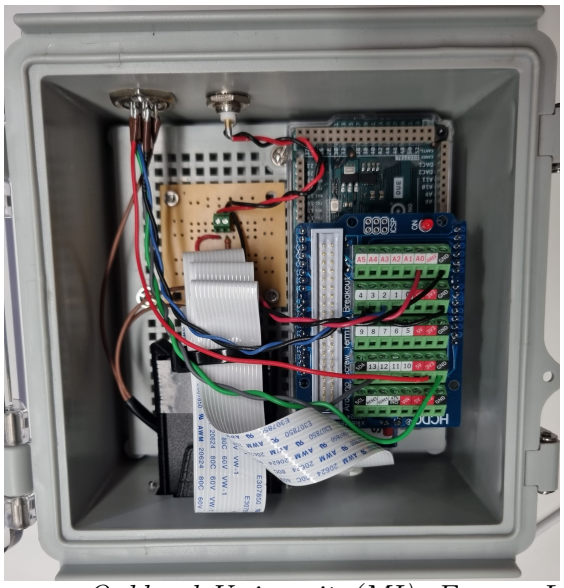
Source: Oakland University(MI), Energy Lab

Once the Encoder setup is settled, it is possible to move on and start working with the DAQ enclosure.

3.1.3 Arduino Container

To achieve a straightforward set-up and removal of the DAQ system from the ICE, all the necessary components are enclosed in a casing, as depicted in Figure 3.4.

Figure 3.4: DAQ enclosure



Source: Oakland University(MI), Energy Lab

A comprehensive list of these components is outlined below:

- Arduino DUE

- Adafruit Data Logger Shield

- Screw terminal Shield

- SD card extender and support

- Voltage Divider

- BNC connector for the pressure transducer

- 6-Pin connector for the encoder

This configuration facilitates the connection of the DAQ, and it simplifies the process of data collection from the SD card as it is accessible from outside the case.

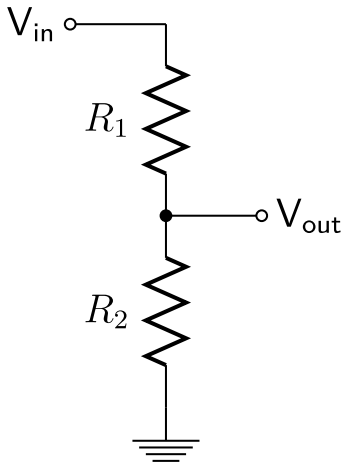
3.2 Electrical Design

This section provides an explanation of the electrical wiring connections, allowing for a better understanding of the general connections from the encoder and the charge amplifier to the Arduino.

3.2.1 Charge amplifier and Voltage Divider

As previously stated in Section 2.3, the pressure transducer's output is within a range of 0 to 10V. To enable voltage readings with the Arduino DUE, it is necessary to decrease the output voltage of the charge amplifier to a range of 0 to 3.3V. This can be achieved through the implementation of a resistive voltage divider.

Figure 3.5: Voltage divider



Source: Wikipedia

Regarding the implementation of the voltage divider in our application, Figure 3.5 displays its configuration where the output voltage of the charge amplifier is used as the input voltage V_{in}, while the output voltage V_{out} is connected and applied to the Arduino DUE. For the resistance aspect, the voltage divider is constructed using a pair of resistors, R_1 and R_2, which are arranged in series, and with R_2 connected to the ground. This arrangement results in the input voltage being divided between the two resistors, producing a voltage output that is proportional to the ratio of the two resistor values.

$$R_1 = 33k\Omega + 33k\Omega \tag{3.1}$$

$$R_2 = 33k\Omega \tag{3.2}$$

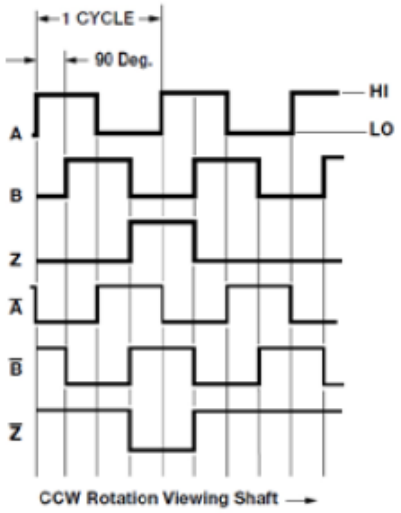
$$V_{out} = \frac{R_2 V_{in}}{R_1 + R_2} = \frac{V_{in}}{3} \tag{3.3}$$

Being R_1 twice R_2 the final, the final V_{out} corresponds to $\frac{1}{3}$ of voltage coming from the Charge Amplifier.

3.2.2 Encoder

The Encoder used in this research is a BEI Sensata XH25D-SS-720-ABZC-4469-LED. It is an Incremental Encoder that has 2 channels in quadrature, 1/2 cycle index gated with negative B channel[7]. The output waveforms are shown in Figure 3.6: The square waves employed for research purposes include the incremental

Figure 3.6: Wave-forms of the Encoder



Source:Sensata Datasheet

B Pulse and the absolute Z Pulse. To gather pressure measurements, only the rising aspect of the signal is utilized. As a result, for every engine rotation, 720 steps will be recorded for the B pulse, which occurs once every 0.5^o CA, while the Z pulse occurs once every 360^o CA. The utilization of an Absolute encoder facilitates system configuration to determine the position of the TDC each time the engine is started. As evident from Figure 3.4, the necessary cables to establish the Encoder's functionality are the 3.3V cable, the Ground cable, B Pulse, and Z Pulse. Furthermore, to minimize the noise in the Encoder output signal, the case ground of the Encoder has also been connected to the Arduino Ground.

3.2.3 Arduino Cable Connections

In this last paragraph, an explanation of the connection between the Arduino DUE, the Adafruit Data Logger Shield, and the Screw terminal Shield are discussed. The Adafruit Data Logger needs to be mounted on top of the DUE with some Shield Stacking Headers, this allows to ensure a strong connection between the Arduino and the SD Shield. These Headers need to be soldered on the Shield PCB otherwise the mechanical connection would not be strong enough to make the system work properly. The last connection that needs to be soldered between the Arduino DUE and the SD Card Shield is the 6 Pin header which needs to be fixed because it is used by the Arduino board to communicate through the SPI protocol with the SD card installed in the Adafruit Data Logger Shield. Once the SD Card shield has been installed it is possible to install the Screw terminal which slides directly into the previously installed Shield Stacking Headers. The Screw Terminal's main function is to connect easily the various inputs and outputs to the Arduino keeping them tight enough to be sure that they do not get loose. The electrical connections are shown in Figure 3.4 and explained below.

- **Pin A0 - Voltage divider Output Red**: This connects the positive side of the Voltage divider to the Arduino DUE board. It is 1/3 of the Voltage reading which is possible to get if the reading would occur from the Charge amplifier Output.
- **Pin GND - Voltage Divider Output Black**: This closes the Charge amplifier circuit and allows the Pressure voltage to be read from the Arduino DUE.
- **Pin GND - Encoder Ground Black**: The black cable is necessary for powering the Encoder, as it completes the power circuit.
- **Pin GND - Encoder Ground Green**: The green cable is connected to diminish the amount of noise on the Encoder outputs, which subsequently enhances the quality of the square-wave signals in the B Pulse and Z Pulse connections.
- **3.3V - Encoder V_{in}**: The Arduino DUE has the capacity to supply power up to 800 mA to any connected device. This feature allows for the encoder to be powered directly by the Arduino, which in turn reduces the number of components required to obtain pressure measurements.
- **Pin 2 - Encoder B Pulse Blue**: This cable transmits the B Pulse square-wave signal, which occurs at a frequency of every 0.5 degrees of crankshaft angle (CA). The signal is utilized to gather pressure measurements each time it rises from 0 to 3.3 volts, which is detected directly by the Arduino DUE.

- **Pin13 - Encoder Z Pulse Grey**: This cable transmits the Z Pulse square-wave signal which occurs once during each full revolution of the crankshaft. This signal serves as an absolute reference point, as it consistently rises in the same position between the Encoder and the Encoder shaft.

Once all these electrical connections have been realized, the SD Card extender can be installed in the enclosure to facilitate the SD removal. This makes it possible to collect the data whenever it is necessary, without having to remove the Arduino from the Box where it is fixed.

The last connection that needs to be installed is the micro-USB cable. Arduino DUE has two different micro-USB ports, the Native USB port, and the Programming Port. In this configuration, the Native USB port is used because it allows to realize a faster connection between the Arduino and the computer to which it is connected. This connection is created through the USB protocol which connects directly to the Micocontroller.

Chapter 4

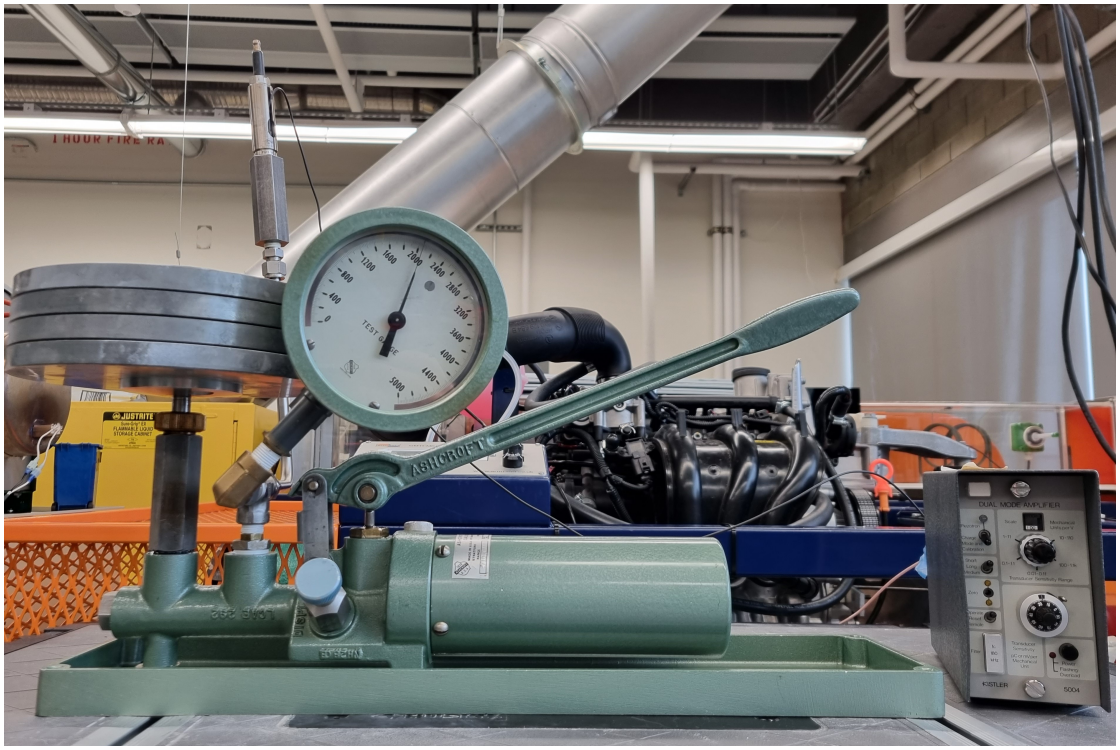
Calibration

This chapter outlines the methodology employed to calibrate two distinct components utilized in this study. Firstly, the Kistler 6118BFD35 Piezoelectric Transducer was calibrated to ensure precise pressure measurements, which were subsequently utilized in post-processing to calculate various parameters associated with combustion. Additionally, the original fuel injectors were calibrated to determine the fuel injector flow rate and the mass of fuel injected as a function of engine speed and injector duty cycle.

4.1 Pressure Transducer

As previously mentioned, the Kistler 6118BFD35 Piezoelectric Pressure Transducer is utilized in the test bench. This sensor operates based on the piezoelectric effect observed in piezoelectric materials like quartz, which generate positive or negative electrical charges when subjected to a mechanical load on the outer surfaces. This charge is produced due to the displacement of positive and negative crystal lattice elements relative to one another, resulting in an electric dipole. The mechanical load on the crystal generates an electrical charge that is directly proportional to the applied pressure[5]. The primary objective of this calibration process is to identify the proportional factor between the voltage measured at a specific pressure. The pressure transducer is typically positioned on the spark plug, which is the most prevalent location for in-cylinder pressure transducers. This is due to the fact that it is readily accessible and can be easily replaced if required. The primary reason for installing the pressure transducer on the spark plug is that there is no need to bore into the top of the cylinder head.

Figure 4.1: Dead Weight Test



Source: Oakland University(MI), Energy Lab

4.1.1 Testing

The Ashcroft Type 1305D Dual Range Deadweight tester(shown in figure 4.1) are precision built primary pressure standards, used for testing, setting, calibrating or repairing pressure measuring devices within the test points 15 psi (100kPa) to 10,000 psi (70,000kPa). The deadweight tester consists of a two stage hydraulic pump containing a manifold which is pressurized during operation. Integral to the pump is a shuttle valve that allows the operator to regulate the speed of pressure increase. One connection to the manifold includes a cylinder and a free-floating precision machined piston with a plate for holding calibrated weights. A second connection to the manifold accommodates a gauge or other pressure measuring device to be calibrated or checked. Incorporated into the manifold is a hand operated displacement valve that allows small adjustments in fluid volume to be made without further operation of the pump handle or release valve. The tester is dual range having two interchangeable piston and cylinder assemblies. One is a low pressure piston having an effective area five times larger than that of the high pressure piston. The low pressure piston is used for making measurements below

2,000 psi (14,000 kPa). The high pressure piston, with an area 1/5 that of the low pressure piston, is used to measure pressure through 10,000 psi (70,000 kPa). The weight masses are pre-measured and identified with the pressure values they produce when operated with the interchangeable piston and cylinder assemblies. Pressure calibration points produced by the deadweight tester are accurate to within ± 0.1% of the reading certified traceable to the N.I.S.T. The tester provides consistent, repeatable accuracy, maintaining its pressure for an appreciable length of time regardless of temperature changes, slight leaks in the pressure system, or changes in volume of the pressurized system due to movement of a Bourdon tube or other device. The theory behind a deadweight tester can be expressed as simply as force acting upon a known area. Pressure produced by the pump is distributed by the manifold, to the base of a precision machined piston and to a device being calibrated or checked. Pre-selected weights loaded onto the piston platform are acted upon by gravity and develop a force that is to be equally opposed by the fluid pressure from the pump. When equilibrium is achieved, the pressure value is known, it being a direct result of the sum of the forces from the weights, piston platform and the piston divided by the effective area of the piston and cylinder assembly [8]. This procedure allows to calibrate the pressure transducer in function of the weights(which apply a corresponding pressure on the piston). To do so, it is important to note down the initial voltage at which no fluid has been pumped into the system, this is needed because a pressure transducer changes its output in function of the variation in pressure, which means that if the pressure stays constant, the voltage output of the charge amplifier starts going down. This calibration is done with the Charge Amplifier with these settings:

- **Scale=10 bar/v**;
- **Range=10-100**;
- **Sensitivity=10.1 pC/bar**;

The charge amplifier output is then sent to the Voltage divider, and the output of the voltage divider is fed to the Arduino DUE in the Analog Input Port that was mentioned in Section 3.2.3. Once these considerations were made, it is necessary to choose the calibration pressures discs which are in psi:

Table 4.1: Calibration Pressures

psi	50	150	350	450	725	950	1150	1450
bar	3.45	10.34	24.13	31.03	49.99	65.50	79.29	99.97

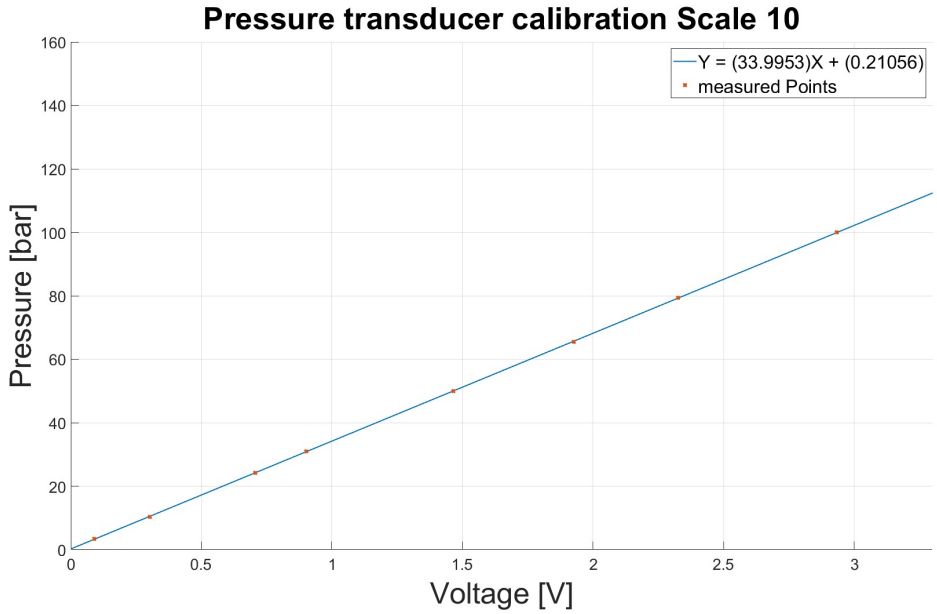
4.1.2 Results

The results of the deadweight test are presented below:

Table 4.2: Calibration Results

ΔV	0.090	0.303	0.706	0.902	1.465	1.926	2.325	2.932
bar	3.45	10.34	24.13	31.03	49.99	65.50	79.29	99.97

Figure 4.2: Pressure Transducer Calibration curve



As it is possible to notice from the results, the relation between the Pressure applied to the Kistler 601A pressure transducer and the output voltage of the charge amplifier is linear with a coefficient:

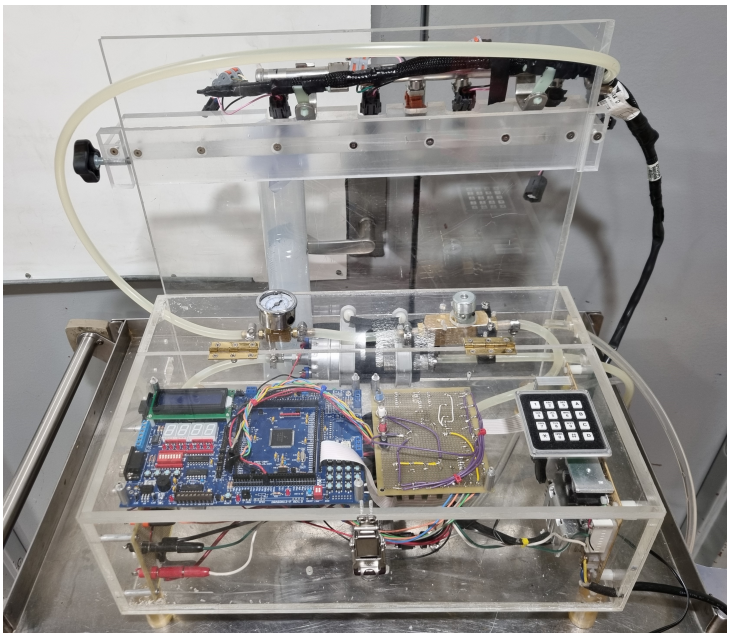
$$C = 33.9953 \frac{bar}{V} \tag{4.1}$$

This allows saying that the calculated coefficient is constant throughout the interval of interest, which helps to have reliable pressure measurements from the voltages which will be collected with this DAQ.

4.2 Fuel Injector

The Armfield CM11- MKII is equipped of a Common rail system for Gasoline Port Fuel injection, it has three injectors, each one of them connected directly to the intake manifold. With the gasoline port fuel injection, the air-fuel mixture is generated outside of the combustion chamber in the intake manifold. The fuel injector sprays the fuel before the inlet valve. During the intake stroke, the mixture flows through the open inlet valve into the combustion chamber. The fuel injectors have been selected such that the fuel demand of the engine is always covered, even at full load and high rotational speed. To conduct the calibration the injectors are electronically controlled, they are mounted in a standard automotive fuel rail. The microcontroller allows to select one of the four fuel injectors which are installed in the common rail, but in this calibration, just three of them are being used. The Period(time between the start of injection pulses), the Pulse Width(injector opening time), and the number of pulses. All these three parameters can be set. The working fluid is stored in a plastic, marine-style fuel tank with a quick disconnect fuel line.

Figure 4.3: Fuel Calibration Front view



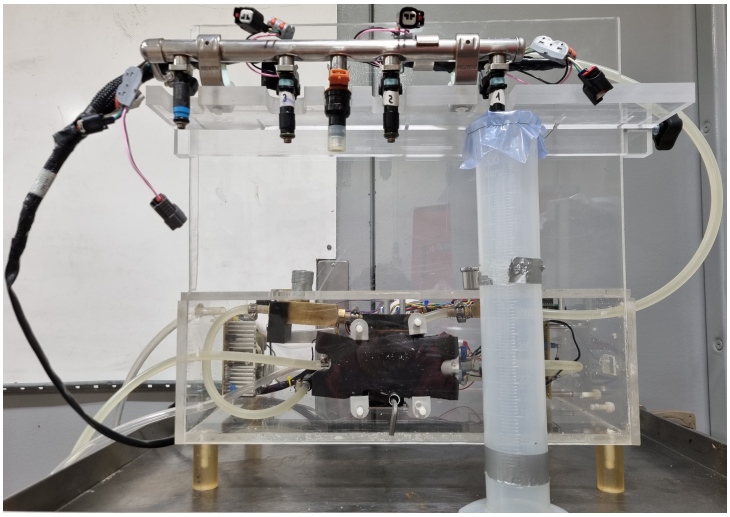
Source Oakland University(MI), Energy Lab

The fluid is pumped, with an aftermarket automotive fuel pump, to the fuel rail and through a regulator back to the fuel tank. The regulator allows adjustment of the fluid pressure to match that typically found in automotive systems (typically

45-60 psi)[9].

In Figure 4.3 it is possible to see the microcontroller which controls the logic that allows to take the test, the number pad which allows to set the previously discussed parameters, the pressure gauge which makes it possible to check if the pressure range utilized to ensure that the pressure range aligns with the specified value of 4 bar for the Armfield CM 11-MKII.

Figure 4.4: Fuel Calibration back view



Source Oakland University(MI), Energy Lab

In Figure 4.4 the back of the fuel calibration testbench is shown, here it is possible to observe the three fuel injectors which need to be calibrated, the common rail at which they are connected, and the graduate cylinder used to collect the volume of fuel which is injected inside the graduate cylinder and then measured in terms of volume.

4.2.1 Testing

The testing process entails selecting an injection period that corresponds to the engine speed and determining the total number of pulses required to obtain a sufficiently large volume for subsequent calculations. The injection period and the number of pulses remain fixed during all measurements, while the Injector Opening Time is the variable parameter that changes with each calibration run. The test conditions are the following:

- $N^o of pulses = 10000$
- Injector Opening timings: 1.7ms, 5.53ms, 9.35ms, 13.18ms, 17ms.

4.2.2 Results

These are the results of the calibration of the fuel injectors:

Table 4.3: Volume injected for 10 000 pulses

Injector Opening Time[ms]	Injector 1 V1[ml]	Injector 2 V2[ml]	Injector 3 V3[ml]
1.7	65	66	66
5.53	170	172	175
9.35	275	280	280
13.18	385	387	389
17	492	495	492

Table 4.3 presents the total volume of fuel injected over 10,000 pulses, which can be used to determine the amount of fuel injected per individual pulse. It is important to note that the injector opening time must exceed a certain minimum threshold in order for fuel to be injected, regardless of whether the opening signal has been sent. This required injector opening time varies among the three injectors:

Table 4.4: Injector minimum opening time

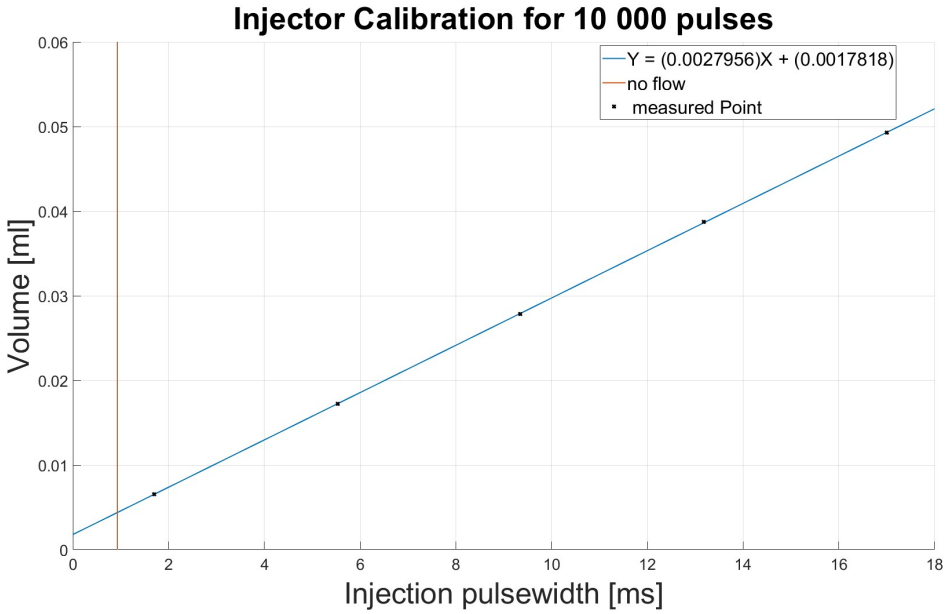
Injector n^o	Required time[ms]
1	0.85
2	0.980
3	0.910

The plot denoting the calibration curve can be observed in Figure 4.5. It is noteworthy that the trend depicted in the curve is linear, implying that the amount of fuel injected is directly proportional to the injector opening time. The vertical red line signifies the minimum opening time of the injector that allows fuel collection. In order to determine the fuel flow, the available injection time needs to be computed, which is contingent upon the engine speed. The duty cycle allows to calculate the amount of time in which the fuel injector is active over the total available time. The process for calculating the injection available time can be derived through a series of steps.

$$K = 2.794 \cdot 10^{-3} \frac{ml}{ms}$$

$$m = 1.7818 \cdot 10^{-3} ml$$

Figure 4.5: Calibration curve



$$t_{available} = \frac{2}{RPM}60 \tag{4.2}$$

$$t_{injection} = t_{available}(inj_{duty}\%) \tag{4.3}$$

$$\dot{V} = N^o cylinder \frac{K \cdot 1000(t_{injection} - t_{required}) + m}{t_{available}} \tag{4.4}$$

with:

- K Fuel flow proportional coefficient
- $t_{available}$ Injector opening time
- $t_{injection}$ Injection time
- inj_{duty} Injector duty cycle
- $\dot{V}$ fuel flow rate
- $t_{required}$ Injector required opening time

Using the density of gasoline it is then possible to calculate the gasoline mass injected per single injection.

Chapter 5

Arduino DUE

An explanation of how data is collected using the Arduino DUE is provided in this chapter. A step-by-step description of the code is made in order to allow a deep understanding of the various functions used to design this data acquisition system.

5.1 Arduino IDE

The Arduino IDE 2.0 is a versatile editor with many features. You can install libraries directly, sync your sketches with Arduino Cloud, debug your sketches and much more. In this section, some of the core features are listed, along with a link to a more detailed article. The Arduino IDE software allows you to:

- **Verify / Upload** - compile and upload your code to your Arduino Board.
- **Select Board & Port** - detected Arduino boards automatically show up here, along with the port number.
- **Sketchbook** - here you will find all of your sketches locally stored on your computer. Additionally, you can sync with the Arduino Cloud, and also obtain your sketches from the online environment.
- **Boards Manager** - browse through Arduino & third party packages that can be installed. For example, using a MKR WiFi 1010 board requires the Arduino SAMD Boards package installed.
- **Library Manager** - browse through thousands of Arduino libraries, made by Arduino & its community.
- **Debugger** - test and debug programs in real-time.
- **Search** - search for keywords in your code.

- **Open Serial Monitor** - opens the Serial Monitor tool, as a new tab in the console.

The sketchbook is where your code files are stored. Arduino sketches are saved as .ino files and must be stored in a folder of the exact name. For example, a sketch named my_sketch.ino must be stored in a folder named my_sketch. With the library manager, you can browse and install thousands of libraries. Libraries are extensions of the Arduino API. The Serial Monitor is a tool that allows the communication with the Arduino Board if it's connected through USB connection. It allows sending commands to the Board and receiving communications when the tasks are completed. The Arduino IDE 2.0 is an open-source project that is free for anyone to download [10].

5.2 DAQ code

In this section, the code of the Data Acquisition system is written in its completeness in order to facilitate the understanding of it. In this way, it is possible to configure a new Arduino DUE and use it to acquire in-cylinder pressure measurements freely.

5.2.1 Main Code

```
#include "SdFat.h"
#include "sdios.h"
#include "Queue.h"
//ncycle is the total number of cycles that we want to record
#define ncycle 100
// Number of analog readings we want to use to make the average
#define nreadings 1

//cyclduration is the number of samples that we will collect for 2 revolutions of the crankshaft
#define cyclduration 1440
#define maxlength 28800  //36000 should be 72 000 Byte of stored data
#define nlimit ((ncycle + 1) * cyclduration)

//Pin where we have the reading of the Voltage from the pressure transducer
```

```
15 #define Voltage_pin A0
16 #define interruptPinZ 13
17 #define interruptPinB 2
18
19
20 const uint8_t SD_CS_PIN = 10;
21 //Enable Fast SPI
22 #define SPI_CLOCK SD_SCK_MHZ(40)
23 #define SD_CONFIG SdSpiConfig(SD_CS_PIN, DEDICATED_SPI,
      SPI_CLOCK)
24
25 //This initializes the object file in which we are going
       to save the data
26 SdFat SD;
27 SdFile dataFile;
28
29 char filename[] = "LOG00.bin";
30
31 //Encoder variables
32 bool flag = LOW;
33 uint16_t CountRevs = 0;
34 int n = 0;            // Counter of the total iterations
      that are being stored
35 uint16_t begin = 0;  //This is used not to reinitialize
      the SD card every time we need to collect data
36 uint16_t ToSD[1];
37
38 Queue<uint16_t> Voltage_readings = Queue<uint16_t>(
      maxlength);
39 const int spispeed = 84000000;
40 uint16_t Atotal = 0;
41 char input = 0;  //This is needed to start the
      acquisition of the data
42
43
44 void setup() {
45   SPI.setClockDivider(spispeed);
46   //analogReadResolution(12);
47
48   if (begin == 0) {
49     SerialUSB.begin(115200);
```

```
50
51
52      while (!SerialUSB) {
53        ;  // wait for serial port to connect. Needed for
    native USB port only
54      }
55      pinMode(10, OUTPUT);  //needed to be sure that the
    chip select pin is in output mode
56      //pinMode(7, OUTPUT); //This is used for test
    purpose only
57      //digitalWrite(7,HIGH);
58      //digitalWrite(7,LOW);
59
60      if (!SD.begin(SD_CONFIG)) {
61        SerialUSB.println("Card failed, or not present");
62        SerialUSB.println("Insert SD card and Restart the
    Arduino");
63        while (1) {
64          ;
65        }
66        // don't do anything more:
67      } else {
68
69        SerialUSB.println("SD card initialized.");
70        SerialUSB.println(" ");
71      }
72      //Setup of the registers for the Direct Access
    memory readings https://youtu.be/XUUI0SCR_cU
73      ADC->ADC_MR |= 0x80;  // these lines set free
    running mode on adc 7 (pin A0)
74      ADC->ADC_CR = 0x02;
75      ADC->ADC_CHER = 0x80;
76      begin++;
77    }
78
79    //file creation
80    if (input != 'y') {
81      SerialUSB.println("Press y to start collecting data:
    ");
82      while (SerialUSB.available() == 0) {}
83      input = SerialUSB.read();
```

```
84      SerialUSB.println(input);
85
86      if (input == 'y') {
87        for (uint8_t i = 1; i < 100; i++) {
88
89          filename[3] = i / 10 + '0';
90          filename[4] = i % 10 + '0';
91          if (!SD.exists(filename)) {
92            //Creates a file if it doesn't exist with that
     name
93            dataFile.open(filename, O_WRITE | O_APPEND |
    O_CREAT);
94            break;
95          }
96          if (i == 99) {
97            SerialUSB.println("Remove files from SD Card")
    ;
98            while (1) {};
99          }
100       }
101       //This condition checks if the file has been
    created
102       if (!dataFile) {
103         SerialUSB.println("Couldn't create file");
104       }
105
106       SerialUSB.print("Logging to: ");
107       SerialUSB.println(filename);
108       dataFile.truncate();
109       Voltage_readings.clear();
110       CountRevs = 0;
111       n = 0;
112
113       attachInterrupt(digitalPinToInterrupt(
    interruptPinZ), readZ, RISING);
114       attachInterrupt(digitalPinToInterrupt(
    interruptPinB), readB, RISING);
115     } else {
116       //This is needed in order to get back to the setup
     if i'm not inserting the right letter
117       begin = 0;
```

```
118         input = 0;
119         setup();
120       }
121     }
122 }
123 
124 void loop() {
125 
126   while (flag == HIGH) {
127 
128 
129     while (Voltage_readings.count() > 0) {
130       ToSD[0] = Voltage_readings.pop();
131       dataFile.write(ToSD, sizeof(ToSD));
132     }
133     if (Voltage_readings.count() > maxlength) {
134       SerialUSB.print("The Queue is full");
135       setup();
136     }
137 
138 
139 
140     if (n == nlimit) {
141       while (Voltage_readings.count() > 0) {
142         //Queue data
143         ToSD[0] = Voltage_readings.pop();
144         dataFile.write(ToSD, sizeof(ToSD));
145       }
146       dataFile.close();
147       SerialUSB.print(ncycle);
148       SerialUSB.println(" Cycle saved");
149       SerialUSB.println("SDcard file Closed");
150       SerialUSB.print("Last Revolution was:");
151       SerialUSB.println(CountRevs - 2);  //-2
    initialization of the vector
152       SerialUSB.print("Last Step was:");
153       SerialUSB.println(n - 1);
154       SerialUSB.println(" ");
155       //Removes all the voltage readings left in the
    Queue
156       Voltage_readings.clear();
```

```
157
158         flag = LOW;
159         input = 0;
160         delay(2000);
161         rstc_start_software_reset(RSTC);  // resets the
    board
162       }
163     }
164 }
165
166
167 void readZ() {
168   flag = HIGH;
169   CountRevs++;  //Increases the value of CountRevs every
      time the Zpulse happens
170 }
171
172 void readB() {
173
174   if (flag == HIGH) {
175     //Average of nreadings pressure readings in order to
      get a more accurate value
176     for (int j = 0; j < nreadings; j++) {
177       while ((ADC->ADC_ISR & 0x80) != 0x80)
178         ;
179       Atotal = Atotal + ADC->ADC_CDR[7];
180     }
181     //https://github.com/sdesalas/Arduino-Queue.h#queueh
    .push(at);
182     //Queue insertion of the voltage value
183     Voltage_readings.push(Atotal);
184     Atotal = 0;
185     if (n == nlimit) {
186       //In order to finish storing the acquired data,
    once the total number of measurements has been
    reached, the interrupts need to be stopped.
187       detachInterrupt(digitalPinToInterrupt(
    interruptPinB));
188       detachInterrupt(digitalPinToInterrupt(
    interruptPinZ));
189     }
```

```
190     n++;
191   }
192 }
```

5.2.2 Queue.h Code

This Queue.h[1] is a library that can be found online but it is not possible to download it from the Arduino library manager.

```
1  #ifndef ARDUINO_QUEUE_H
2  #define ARDUINO_QUEUE_H
3
4  #include <Arduino.h>
5
6  template<class T>
7  class Queue {
8    private:
9      int _front, _back, _count;
10     T *_data;
11     int _maxitems;
12   public:
13     Queue(int maxitems = 256) {
14       _front = 0;
15       _back = 0;
16       _count = 0;
17       _maxitems = maxitems;
18       _data = new T[maxitems + 1];
19     }
20     ~Queue() {
21       delete[] _data;
22     }
23     inline int count();
24     inline int front();
25     inline int back();
26     void push(const T &item);
27     T peek();
```

[1]This code is open source. Can be downloaded from the Internet and implemented in your code. The library is not available yet. Source:https://github.com/sdesalas/Arduino-Queue.h

```
28      T pop();
29      void clear();
30  };
31
32  template<class T>
33  inline int Queue<T>::count()
34  {
35    return _count;
36  }
37
38  template<class T>
39  inline int Queue<T>::front()
40  {
41    return _front;
42  }
43
44  template<class T>
45  inline int Queue<T>::back()
46  {
47    return _back;
48  }
49
50  template<class T>
51  void Queue<T>::push(const T &item)
52  {
53    if(_count < _maxitems) { // Drops out when full
54      _data[_back++]=item;
55      ++_count;
56      // Check wrap around
57      if (_back > _maxitems)
58        _back -= (_maxitems + 1);
59    }
60  }
61
62  template<class T>
63  T Queue<T>::pop() {
64    if(_count <= 0) return T(); // Returns empty
65    else {
66      T result = _data[_front];
67      _front++;
68      --_count;
```

```
69      // Check wrap around
70      if (_front > _maxitems)
71        _front -= (_maxitems + 1);
72      return result;
73    }
74  }
75
76  template<class T>
77  T Queue<T>::peek() {
78    if(_count <= 0) return T(); // Returns empty
79    else return _data[_front];
80  }
81
82  template<class T>
83  void Queue<T>::clear()
84  {
85    _front = _back;
86    _count = 0;
87  }
88
89  #endif
```

5.3 Code explanation

In this section, the different functions used to develop this Data Acquisition System are described in detail.

5.3.1 Arduino base commands

The code for Arduino is typically organized into three primary sections. The first section is known as the initialization stage, where constants and variables are defined. When variables are defined in this section, they become global variables that can be accessed from any other part of the code.

The second section of the code comprises:

```
1     setup(){
2     //setup code is written here
3
4     }
```

The setup() function is called when a sketch starts. The setup() function will only run once, after each powerup or reset of the Arduino board[11]. It can be used for actions that need to be taken just once, it is possible to recall the setup() function from the code allowing it to run again. The third part of the code is the:

```
1    loop(){
2    //This part of the code runs continuously
3
4    }
```

The loop() function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board[12]. Outside of the loop() function it is possible to have other functions which can improve the Arduino's Capabilities. Once the sketch layout has been explained, the syntax used to develop this Data Acquisition System is described. Starting from the beginning of the code the libraries which are needed for the DAQ are declared with the function:

```
1    #include "library_name"
```

This line is used to include outside libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino[13]. Three different Libraries need to be included for this application, **SdFat.h** and **sdios.h** are related to the SD Card library which allows to work with FAT16/FAT32 and exFAT SD card, the third library which is **Queueu.h** includes a circular queue for Arduino embedded projects. After including the different libraries, the constant parameters are defined using this syntax:

```
1    #define parameter_name parameter_value
```

It is a useful C++ component that allows the programmer to give a name to a constant value before the program is compiled. Defined constants in Arduino don't take up any program memory space on the chip. The compiler will replace references to these constants with the defined value at compile time.

This can have some unwanted side effects though if for example, a constant name that had been defined is included in some other constant or variable name. In that case the text would be replaced by the # defined number (or text) [14]. Defining constants allows to optimize the code where values do not change throughout

the running code. To be able to use the variables they need to be declared, the assignation is done with a declaration:

```
1    type_var name_var=value_var;
```

The main types of variables used in this code are uint16_t, int, char, and bool, respectively:

- **uint16_t** defines an unsigned positive integer number $[0;(2^{16})-1]$[15]
- **int** stores a signed 32-bit (4-byte) value.$[-2^{31}; (2^{31})-1]$ [16]
- **char** is used to store single characters[17]
- **bool** holds one of two values, true or false[18]

The second part of the code is now explained and a description of the setup() function is written.

```
1    SPI.setClockDivider(spispeed);
```

The system clock can be divided by values from 1 to 255. The default value is 21, which sets the clock to 4 MHz like other Arduino boards.[19] It is possible to find the Serial function, which is used for communication between the Arduino board and a computer or other devices.

```
1    SerialUSB.begin(baudrate);
2    SerialUSB.println("text to print");
3    while (SerialUSB.available() == 0);
4    input = SerialUSB.read();
```

Line 1 sets the data rate in bits per second (baud) for serial data transmission to a Serial Monitor[20]. Line 2 Prints data to the serial port as human-readable ASCII text followed by a carriage return character (ASCII 13) and a newline character (ASCII 10)[21]. Line 3 instructs the Arduino to wait until a SerialUSB connection is established between the Arduino and the computer. Finally, on Line 4, incoming serial data is read and assigned to a variable.

```
1    rstc_start_software_reset(RSTC); //resets the board
```

This line has the same functionality as the Reset button on the board but it allows to do it via software. Pressing it has the same effect as disconnecting and reconnecting the power supply, it will start executing any instructions in the sketch from the beginning. Powering down the board clears RAM memory, so values that were previously assigned to variables are not kept [22].

After this small explanation about the main functions which can be used to write code for Arduino, a more specific explanation is done with respect to the functions and libraries that are used for this application.

5.3.2 External interrupts

It is important to connect the Encoder outputs to the pins which are capable of the interrupt function. On the Arduino DUE it is possible to set the external interrupts in all the digital pins. Interrupts are useful for making things happen automatically in microcontroller programs and can help solve timing problems. Good tasks for using an interrupt may include reading a rotary encoder, or monitoring user input. If you wanted to ensure that a program always caught the pulses from a rotary encoder, so that it never misses a pulse. ISRs are special kinds of functions that have some unique limitations most other functions do not have. An ISR cannot have any parameters, and it shouldn't return anything. If your sketch uses multiple ISRs, only one can run at a time, other interrupts will be executed after the current one finishes. Typically global variables are used to pass data between an ISR and the main program[23]. The external interrupt functions needs to be activated for each output of the encoder. The command for calling the external interrupt is:

```
1    attachInterrupt(digitalPinToInterrupt(pin), ISR,
   mode);
```

Once this function is called the external interrupts are working. In the function definition it is needed to set the different parameters:

- Pin depends on the number where the output pulses are connected.
- The B pulse is connected to pin n^o 2 on the Arduino Due, while the Z pulse is connected to pin n^o 13. ISR is the name of the function which runs when the two different square wave outputs occurs.
- Mode can have different triggers which activate the code written in the function connected to the interrupt. In this case the RISING of the signal in order to have a trigger every time the Encoder output state goes from low to High state.

The utilization of the external interrupts allows the loop code to run continuously while waiting for the next interrupt. In this way the software is always running, and once the the Z Pulse occurs, an increment on the revolution counter is done. While the Z Pulse occurs once every cycle, the B Pulse occurs once every 0.5^o CA, when this happens the analog output of the pressure transducer is collected from the ADC and is stored in an Arduino variable. In the next paragraph the Analog to digital conversion is explained. To be sure that the data collection stops when the total number of measurement points is obtained it is necessary to use the function:

```
1    detachInterrupt(digitalPinToInterrupt(pin));
```

This function detaches the external interrupts which means that even if there is a square wave signal going to the Arduino digital pins, no operation runs from the ISR. This allows to finish running the loop() code without interrupting the loop() routine which can be defined with a low-priority code.

5.3.3 Analog input

The Arduino DUE has an integrated ADC system that allows to convert of an analog input that goes from 0 to 3.3V to a digital number that goes from 0 to 4095. This means that in order to have the corresponding tension value, a conversion needs to be done.

$$V = \text{measure}_{\text{value}} \frac{3.3}{4095} \tag{5.1}$$

Different methods of reading an analog voltage can be set up on the Arduino DUE. The standard Arduino method is written below:

```
1    pinMode(A0,INPUT);
2    Reading=analogRead(pin);
```

Line 1 is needed the Pin A0 in INPUT mode, it is necessary otherwise no output is read from the analogRead() function. When Line 2 is called, the ADC is activated, reads the voltage from the output, and assigns to the Reading variable a number that goes from 0 to 4095. Using the analogRead() function takes about 7.3 μs. The total available time at 4000 RPM can be calculated using the Formula 2.1 that becomes:

$$\text{Sampling Rate} = 2 \left(\frac{[4000]}{60 \frac{[\text{sec}]}{[\text{min}]}} \right) 720 \text{Sampling Rate} = 96000 Samples/s \tag{5.2}$$

In this way it is possible to calculate the sample time which is:

$$\text{Sampling Time} = \frac{1}{\text{Sampling Rate}} \tag{5.3}$$

$$\text{Sampling Time} = 10\mu s \tag{5.4}$$

This means that on the leftover time, it would be necessary to store the acquired variables in the SD Card. This means that the risk of losing data is very high and this makes drives to investigate faster analogRead methods. The other solution is Direct Access Memory. To do so, the Datasheet needs to be studied, working directly with the ADC means that pointers need to be used to assign values to the registers.

```
1    ADC->ADC_MR |= 0x80;
2    ADC->ADC_CR = 0x02;
3    ADC->ADC_CHER = 0x80;
```

Line 1 selects the free running mode for the ADC which means that the Arduino Analog to Digital conversion is running continuously while the Arduino is doing other operations. Line 2 is the Control register which starts the conversion, which means that the Arduino is continuously converting from Analog to Digital. Line 3 is the Channel Enable register which is needed to select the Pin at which the Analog input is inserted. Two lines of code allow to sample the ADC:

```
1    while ((ADC -> ADC_ISR & 0x80)) != 0):
2    Reading = ADC -> ADC_CDR[7];
```

The first line checks if there is a data point ready, and if there is not, the software waits until a Voltage reading is converted to an equivalent 0 to 4095 number. The Interrupt Status Register allows to know when the ADC has completed a conversion, this means that this line of code allows the system to wait if there is no converted value yet. Line 2 is the assignation to a variable of the value which resides at the ADC collected, it is a read-only register inside the ADC and corresponds only to the channels which are enabled[24]. Using this method gives a sampling time that is about $0.6\mu s$ due to the fact that the ADC is always running even when the data is not required yet but the Arduino stores it only when the encoder triggers the B Pulse.

5.3.4 SD Card implementation

To store the data it is necessary to integrate to this DAQ an external memory where to save the measurements. This is required because the onboard Flash

memory of the Arduino DUE is not usable, while the SRAM capacity is not big enough to store 100 cycles. For this reason, the introduction of an SD Card in this Data Acquisition System is necessary. In this research, the Adafruit Data Logger Shield is used, it is compatible with the **Sdfat.h** library which is the fastest SD library available at the moment. Starting from the first part of the code, the first parameter that needs to be set is:

```
1     const uint8_t SD_CS_PIN = 10;
```

This assignment needs to be done to allow the microcontroller to work with the Adafruit Shield and communicate with the SD Card.

```
1     #define SPI_CLOCK SD_SCK_MHZ(40)
2     #define SD_CONFIG SdSpiConfig(SD_CS_PIN,
    DEDICATED_SPI, SPI_CLOCK)
```

The first line sets the SPI speed at which the communication between the Arduino and the SD Card is going to run. The second line configures the SD Card through the setting of the communication pin which changes if another shield is used. The dedicated SPI is set due to the fact that no other SPI devices are connected to the Arduino and this allows to run the interface at the maximum speed. The last parameter is the SPI clock speed which was defined in Line 1

```
1     SdFat SD;
2     SdFile dataFile;
```

The first line initializes SD object while the second one defines the file object which will be used to call the various functions that allow the management of the files and also to save the parameters needed to the SD Card.

```
1 if (!SD.begin(SD_CONFIG)) {
2       SerialUSB.println("Card failed, or not present");
3       SerialUSB.println("Insert SD card and Restart the
    Arduino");
4       while (1) {
5         ;
6       }
```

This part of the code verifies if the SD Card is present in the Adafruit Shield and if it is present, it starts the communication between the Arduino and it. If the card is not present or there is an error in the communication then the Arduino prints out the error and enters an infinite loop, which can be reset only by disconnecting and reconnecting the Arduino to the computer, or pressing the reset button on the PCB. This is made by choice so that while running the test, it is possible to notice that the SD card is missing.

```
1     if (!SD.exists(filename))
```

This line of code checks if there is a file inside the SD card with the exact name which is associated with the filename variable. If there is no file associated with it, it goes on with the next line of code:

```
1     dataFile.open(filename, O_WRITE | O_APPEND | O_CREAT
    );
```

This line of code creates a file with the filename associated with the stored variable name. If the file is already present, it continues adding the values that we are sending to it physically at the end of the file.

```
1       dataFile.truncate();
```

This .truncate() function helps to speed up the storage of the data on the SD card, this happens because the Arduino truncates the file from one position on which means that the Arduino needs to save the data in an empty space. Finally the last command of the Arduino Library is called:

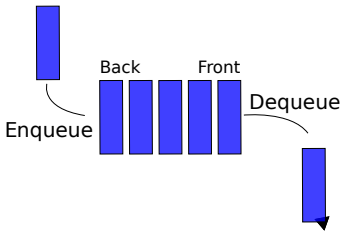
```
1     dataFile.write(ToSD, sizeof(ToSD));
```

This command is used to send the data to the SD card, the data which are going to the SD is written in bytes, and sizeof(ToSD) is needed to specify the number of bytes that are being sent to the SD card. The code which is writing the measured data to the SD card is running in the main loop, which means that compared to the ISR code has a lower priority. This is done because the SD card works with an internal buffer and when this buffer is full, it can take up to 160μs which means that running everything with the same priority would cause some data losses while normally when the buffer is not full, it takes up to 6 μs.

5.3.5 Queue implementation

A Queue is required in this code to enable data collection while the interrupts triggered by the B pulse are in progress. During the execution of the interrupt code, writing to the SD Card is suspended to allow the Arduino to locally store the measurements on the SRAM. The interrupts are executed with higher priority than the main loop function, facilitating the collection of data as soon as the B pulse is detected by the Arduino without any loss of critical data. This system allows transmission of data to the SD Card when the Arduino is free to execute the loop function. For this purpose, the measurements are temporarily stored in a Queue and subsequently transmitted to the SD Card using the Queue library. **Queue.h** is a circular queue for Arduino embedded projects, it follows the FIFO which means that the First element that enters the queue is the first element that gets extracted.

Figure 5.1: Queue Schematics



Source: Wikipedia

Figure 5.1 represents the logic that is used by the library. The head is the first element that is inserted into the queue, when an element needs to be inserted, it is connected to the back of the queue. This operation makes it possible to keep the list ordered, when an element needs to be removed from the list, it is taken off from the front of the Queue. These are the main operations that are needed to work with the queue. At this point, the code explanation is shown.

```
1    #define maxlength 28800
2    Queue<uint16_t> Voltage_readings = Queue<uint16_t>(
   maxlength);
```

This line defines the structure of the queue, it is declared as a queue of uint16_t elements and it can reach a maximum length of 28800 elements, this is necessary because when a high sampling rate is required, it is necessary that the measured voltages need to be collected on the back of the queue, while the front of the queue is sent to the SD card.

```
1    Voltage_readings.clear();
2    Voltage_readings.count()
```

Line 1 is the function that removes all the elements from the queue while Line 2 counts how many elements are present inside the queue, this is needed because a check on the number of measurements which are already inside is needed. If the number of elements in the queue reaches the maximum it means that some data may be lost, for this reason, an error has to be set.

```
1    Voltage_readings.push(Atotal);
2    ToSD[0] = Voltage_readings.pop();
```

The last two commands used from this **Queue.h** library are these ones. Line 1 adds a generic element at the back of the queue. In this case, it is used to store the analog voltage which is read from the A0 pin.

Line 2 gets a generic item from the front of the queue, which in this case is assigned to a variable that sends the value to the SD card.

5.3.6 Data output

To keep up with the amount of data that needs to be stored at the highest speeds some considerations need to be done. First of all the amount of data out needs to be reduced as much as possible, this means that just the pressure measurements can be stored, not saving on the SD Card any other parameter that can be measured with the Arduino. For example, there is the possibility to read the number of revolutions and the total number of steps, but this would slow down the process of saving the data to the SD card. The second observation which has to be made is about the format of the numbers which are sent to the SD card, using the command:

```
1    dataFile.write(ToSD, sizeof(ToSD));
```

This command writes to the SD card the value corresponding to the variable in function of how many bytes the variable has. In this way, only the useful bytes are going to be written on the SD card.

Chapter 6

MATLAB Code

Once the data are collected, it is possible to open them with a MATLAB function, to do so, first of all, it is necessary to convert the file from a Binary code to a Human readable number, then it is possible to make some post-processing on the code, in order to find the Pressure curve in function of the Crank Angle and so on.

6.1 Data Opening

To be able to read the data acquired from the Arduino, the file needs to be opened directly from MATLAB because it is saved in the .bin file.

```
1 fileID = fopen("LOG17.bin");
2 A= fread(fileID,[1,inf],'uint16')';
```

Line 1 opens the file, filename, for binary read access, and returns an integer file identifier equal to or greater than 3[25]. This File Identifier is then sent to the function in Line 2, A = fread(fileID,sizeA,precision), the fread function reads file data into an array, A, with dimensions, sizeA, and positions the file pointer after the last value read. fread populates A in column order. Values are interpreted in the file according to the form and size described by precision[**freadmatlab**]. Once the numbers readable, it is needed to convert them from integers to Voltages, the procedure to do so is the one explained in paragraph 5.3.3:

$$V = \text{measure}_{\text{value}} \frac{3.3}{4095} \tag{6.1}$$

These are the voltage output that the Arduino reads in function of the crank angle, to have the pressure measurements another multiplication needs to be done.

6.1.1 Pressure Readings

To be able to convert the voltage measurements that are collected while the engine was running into pressure measurements, the parameter which is calculated in Chapter 4.1.2. It is important to remember that some pressure referencing has to be done in order to have accurate data, this is needed because the piezoelectric pressure transducer is not an absolute pressure transducer but it is a relative one. It is capable to detect the pressure change, but not the absolute value with respect to the external atmospheric pressure.

6.2 Pressure Pegging

By design, transducers respond to pressure differences by outputting a charge referenced to an arbitrary ground. Thus to quantify absolute parameters such as peak pressure and burn durations from heat release analyses, the transducer output must be directly correlated to pressure(pegged) at some point in the cycle. The procedure chosen is to set cylinder pressure at the inlet bottom dead center equal to intake Manifold Absolute Pressure(MAP). This is a very accurate procedure in untuned intake systems or at very low speeds in tuned systems. This method loses some accuracy when the engine speed increases. A second limitation of this procedure is that the pegging is done at a specific point in the cycle at IBDC. Noise in the experimental data would lead to inaccurate referencing for the remainder of that cycle. For this study, an average of five points, one crank degree before and one after IBDC was used to reduce the potential for error associated with relying on a point measurement[26]. In this MATLAB code, the pressure measurements are organized in a matrix with 1440 lines and a number of columns which depends on the number of cycles which are requested to be recorded, it can vary from 100 to 1000 cycles.

```
1 %% Peg Cylinder Pressure to Manifold Pressure Within
     Window near -180deg
2 CApegi = -182;
3 CApegf = -178;
4 P.peg = mean(P.filt(find(CA >= CApegi,1):find(CA >=
     CApegf,1),:),1) - (CAL.Pamb + P.MAP);
5
6 % Repeg Raw and Filtered Pressure Traces to Manifold
     Pressure
7 P.raw = P.raw - P.peg;
8 P.filt = P.filt - P.peg;
```

The average in-cylinder pressure at IBDC is calculated with five points because in this case, our pressure measurements have an increment of 0.5^o CA. This allows us to calculate the pressure shift for every single cycle and corrects every cycle to have an in-cylinder pressure measurement equal to the MAP in this interval. As written before this correction allows to have accurate pressure measurements.

6.3 Pressure Filtering

Low-pass filters are widely used as they are suitable for retaining the physical information useful for combustion analysis while removing high-frequency noise. The main problem associated with low-pass filters is the determination of the optimum cut-off frequency[27]. To filter these pressure data a MATLAB function is used:

```
1 [b,a]=butter(n,Wn,'low');
2 pressure_avg_filt=filtfilt(b,a,pressure_avg);
```

Line 1 returns the transfer function coefficients of an nth-order lowpass digital Butterworth filter with normalized cutoff frequency Wn. The filter order represents Wn is defined as:

$$W_n = \frac{2\mathrm{f}_{\mathrm{cutoff}}}{\frac{RPM}{60}\mathrm{Cycle}_{\mathrm{duration}}} \tag{6.2}$$

The cutoff frequency needs to be tuned in order to have a smooth resultant pressure profile. Line 2 performs zero-phase digital filtering by processing the input data x in both the forward and reverse directions. After filtering the data in the forward direction, the function reverses the filtered sequence and runs it back through the filter. The result has these characteristics:

- Zero phase distortion
- A filter transfer function equal to the squared magnitude of the original filter transfer function
- A filter order that is double the order of the filter specified by b and a

Zero-Phase filtering helps preserve features in a filtered time waveform exactly where they occur in the unfiltered signal. b and a are the transfer function coefficients defined previously. Tuning these parameters it is possible to obtain smooth curves for what regards the pressure transducer.

6.4 Diagrams

In this section different diagrams which are possible to produce using the acquired data are explained.

6.4.1 Pressure signals

The pressure diagrams can be plotted in relation to the crank angle degree, this allows observing the variation in pressure between each cycle. It is possible to plot the filtered pressures superimposed to detect the variability in the combustion cycle. It is also possible to calculate the average pressure with respect to all the measured pressure values. These plots are shown in Chapter 7 divided into different categories, starting from the idle load and idle speed recording 100 cycles, then increasing the engine speed, and load, from 100 stored cycles up to 1000 cycles.

6.4.2 P-V diagram

The pressure-volume diagram is of interest because the work W produced by the cylinder gas is:

$$W = F\Delta x = pA\Delta x + pdV \tag{6.3}$$

Where:

$$F = \text{force acting on the piston}$$

$$\Delta x = \text{incremental distance of the piston movement}$$

$$p = \text{in-cylinder pressure}$$

$$A = \text{piston cross-sectional area}$$

$$\Delta V = \text{incremental volume swept by piston}$$

Thus when pressure is plotted against volume the area under the curve represents the indicated work done by/on the cylinder gas. A shortcoming of the pressure-volume (p-V) diagram is that the greatest rate of change of pressure occurs during combustion near TDC of piston motion, when, given the traditional crank mechanism, the velocity of the piston is low[28]. To calculate the volume, the engine characteristics are required:

$$\text{Number of Cylinders} = 3$$

$$\text{Bore diameter} = 76.5\text{mm}$$

$$\text{Stroke} = 86.9\text{mm}$$

$$\text{Rod length } L = 138\text{mm}$$

$$\mathrm{r} = \frac{S}{2}$$

$$\lambda = \frac{r}{L}$$

$$\mathrm{CR} = 10.3$$

$$V_{clearance} = \frac{2r\pi\frac{B^2}{4}}{CR - 1}$$

$$x(\theta) = r\left[1 - cos(\theta_{rad}) + \frac{1}{\lambda}\left(1 - \sqrt{1 - \lambda^2 sin(\theta_{rad})^2}\right)\right] \tag{6.4}$$

$$V(\theta) = V_{clearance} + \pi\frac{B^2}{4}x(\theta) \tag{6.5}$$

It is possible to calculate the displacement volume as a function of the crank angle, which is required to create the P-V diagram and the indicated cycle.

6.4.3 Indicated Cycle

The indicated cycle is characterized by both axes in the logarithmic scale. Using the logP-logV diagram it is possible to get pressure and volume at intake valve closure and exhaust valve closure, this makes it possible to calculate easily the polytropic exponent:

$$pV^k = const \tag{6.6}$$

$$ln(p) + kln(v) = ln(const)$$

$$k = \frac{log\frac{P_{-100^oCA\ \mathrm{bTDC}}}{P_{-20^oCA\ \mathrm{bTDC}}}}{log\frac{V_{-100^oCA\ \mathrm{bTDC}}}{V_{-20^oCA\ \mathrm{bTDC}}}} \tag{6.7}$$

To calculate the polytropic coefficient, the assumption is made that the valve intake opens at -100 CA degrees and closes at -20 CA degrees, as stated in Equation 6.7. In Gasoline Internal Combustion Engines (ICE), the polytropic coefficient typically falls between 1.32 and 1.4. This coefficient represents the compression phase of the air within the cylinder, accounting for the heat transfer component as well.

Although the theoretical and experimental lines should be coincidental during the compression phase, they may differ slightly due to various parameters. For example, an inaccurate calibration of the clearance volume may result in imprecise estimations of the volume function. In this case, certain engine parameters were obtained from engines with similar maximum power and cylinder displacement, with the assumed parameter being the rod length.

Chapter 7

DAQ Validation

This chapter presents the results acquired with the designed Data Acquisition System discussed in the research. To validate the DAQ the testbench present in the Oakland University Energy Lab is used to acquire data under several working conditions. Different engine speeds and loads have been tested changing also the amount of engine cycle stored continuously from the DAQ.

7.1 Engine speed evaluation

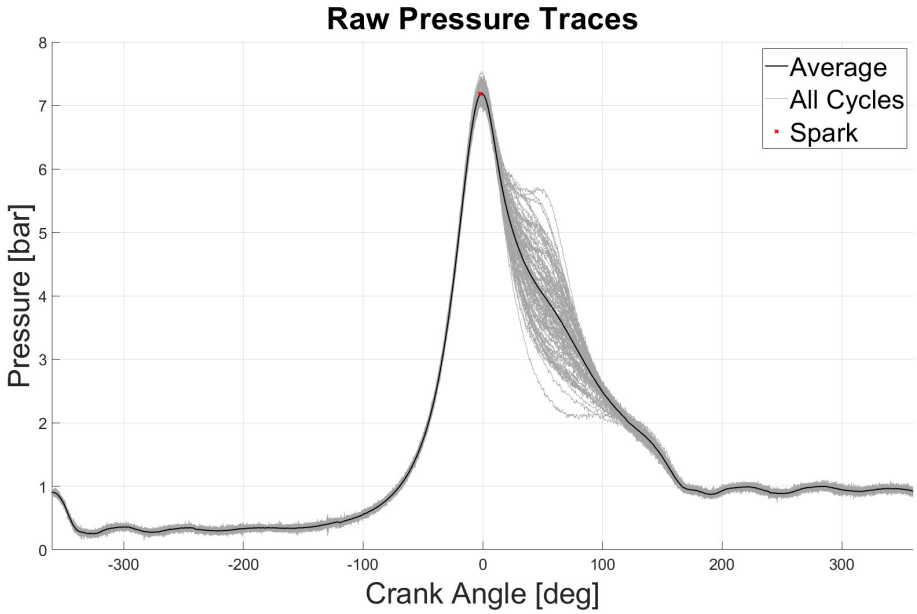
In order to assess the performance of the Arduino DUE-based DAQ, a series of tests were conducted using various engine speeds. The tests commenced at an idle speed and gradually increased to a maximum of 4000 RPM, as detailed in Table 7.1. Each test run stored 100 data cycles.

Table 7.1: Engine speed test conditions

Speed[RPM]	MAP[bar]	Throttle position[%]	Power[kW]
930	-0.67	0	0.20
1500	-0.64	8	1.74
2000	-0.66	9	2.76
2500	-0.70	10	3.09
2800	-0.76	10	2.43
3000	-0.75	11	3.05
3500	-0.77	12	3.11
4000	-0.77	13	1.93

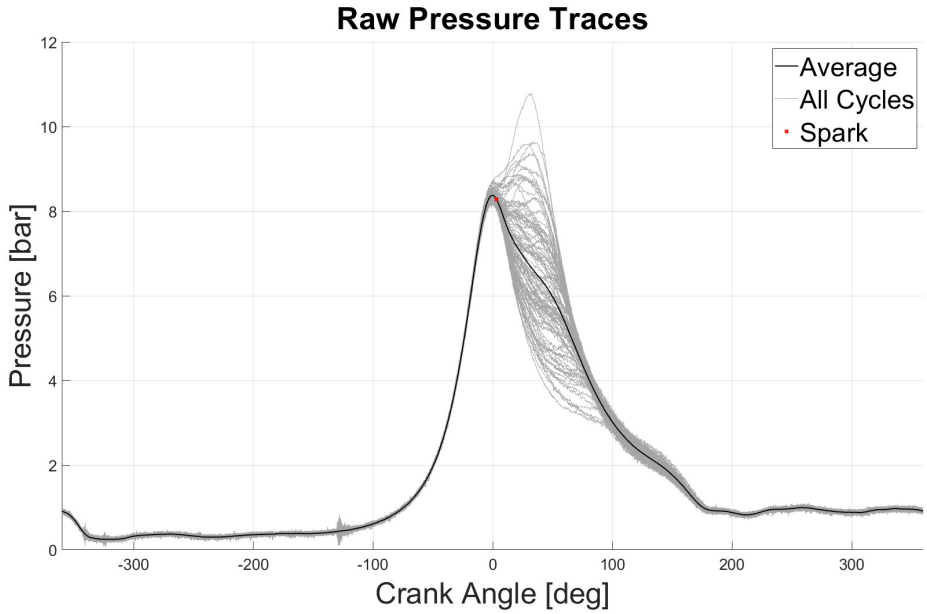
In this section, the plots with the results acquired from the DAQ are presented, in Figure 7.1 the raw pressure signal in function of the Crank Angle Degree at idle speed is presented. The intake phase starts at -360^{o}CA, with TDC at -0.5^{o}CA. It is possible to notice that at idle speed the combustion phase happens after TDC, in fact, the combustion hump is localized after the Top Dead Center. In the figures, the average cycle is represented with a Black line and it is calculated by averaging the pressure measurement for every 0.5^o CA over the 100 cycles which are collected consecutively.

Figure 7.1: 930 RPM Throttle valve 0%



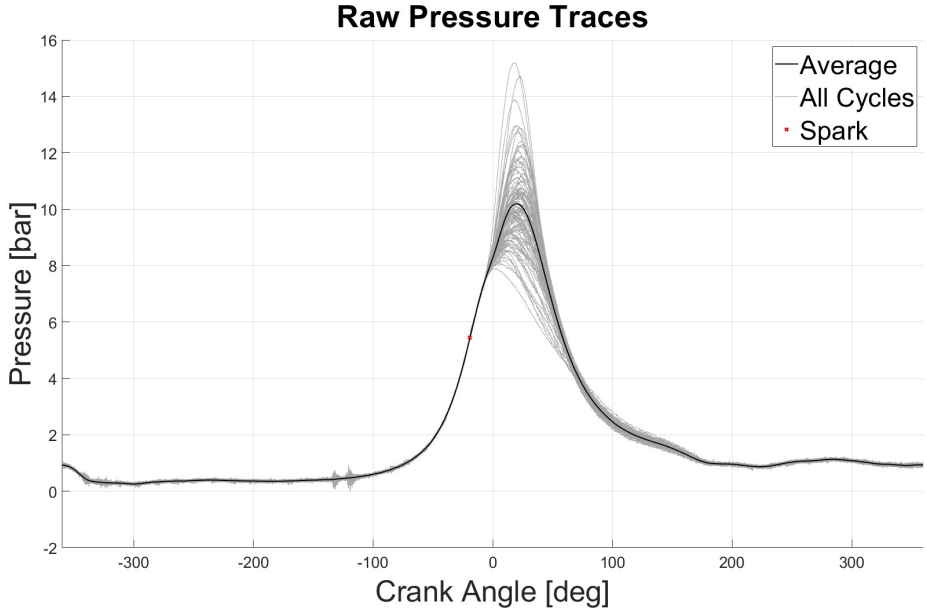
In Figure 7.2 the test is running at low load at a speed of 1500RPM, it is possible to notice that the cycle variability is bigger than it was before, with a peak in pressure which happens slightly after the TDC. This happens because the ECU varies continuously the running conditions in order to keep combustion stable, and the running conditions as requested by the Armfield Software.

Figure 7.2: 1500 RPM Throttle valve 8%



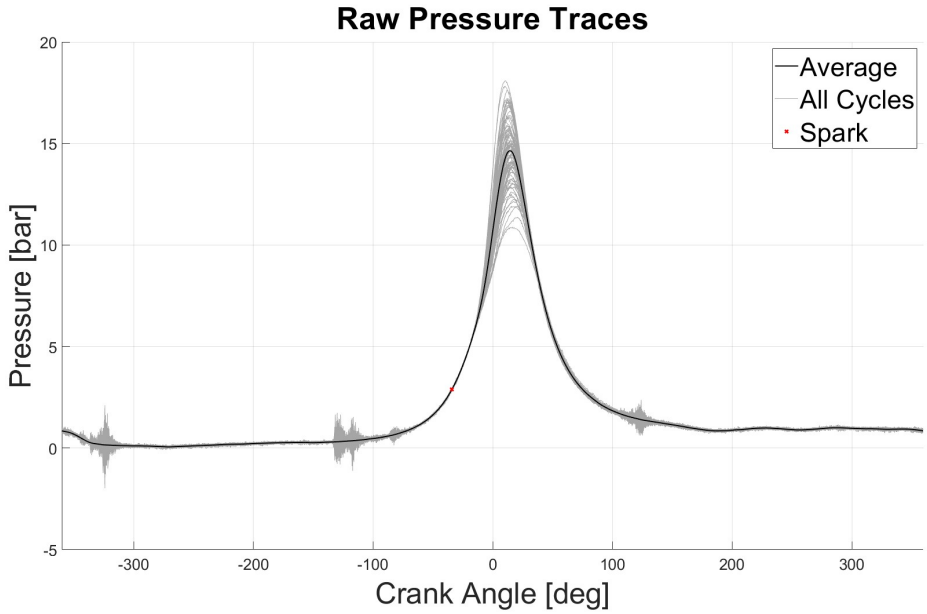
In Figure 7.3 increasing the engine speed even if the engine load is low, increases the pressure peak after TDC. It is noticeable that the pressure signals are clear enough and there is not a lot of noise, what is important to consider is the variability of the pressure in each cycle. The maximum peak in pressure during the combustion phase reaches values of 15 bar for a few cycles with an average value of 11 bar over 100 cycles. In this figure, it is possible to notice some noise high-frequency noise which is likely to be caused by the valve events.

Figure 7.3: 2000 RPM Throttle valve 9%



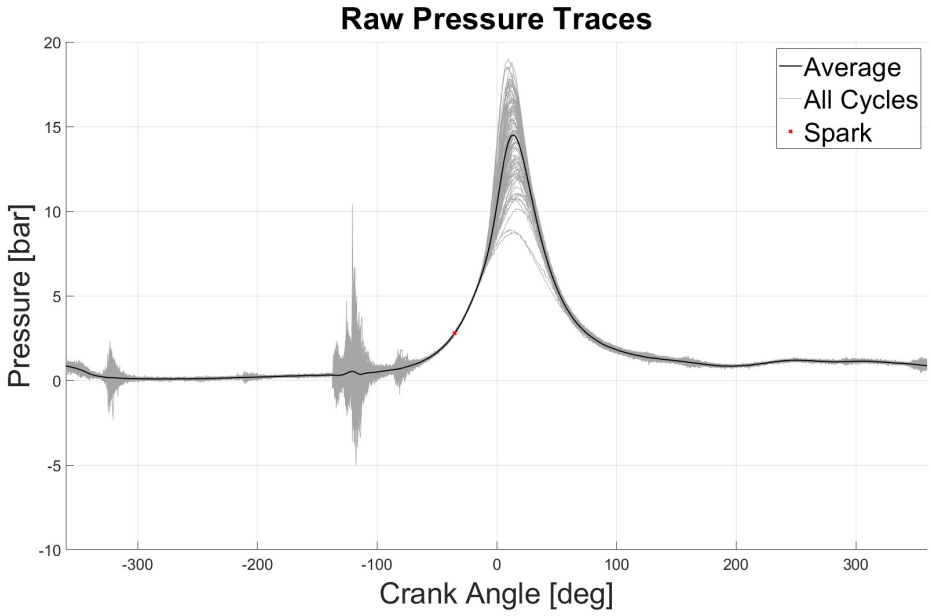
In Figure 7.4 the pressure peak increases even more due to the fact that the throttle valve opening percentage is higher, which means that a larger amount of air enters the cylinder together with a larger amount of fuel. The level of noise increases with respect to the previous measurements, but the noise is showing always at the same points. The peak pressure reaches 18 bar.

Figure 7.4: 3000 RPM Throttle valve 11%



In Figure 7.5 the trend continues, peak pressure increases, and also the pressure traces in the combustion phase are spread more. The noise increases even more, but as it is possible to see later, increasing the engine load, decreases the noise in the pressure trace.

Figure 7.5: 4000 RPM Throttle valve 13%



7.2 Engine Load Evaluation

In this Section, the results of the pressure trace changing the engine load are shown. These pressure measurements are taken considering a fixed engine speed, but increasing the engine load for each measurement.

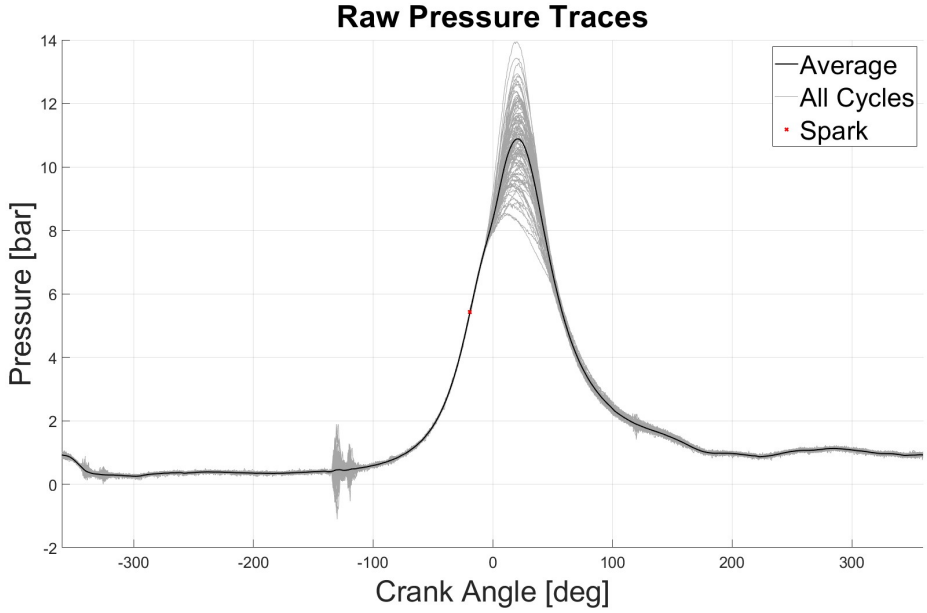
Table 7.2: Engine load test at 2000RPM

Throttle position[%]	Brake load [%]	Power[kW]	MAP[bar]
9	15	2.95	-0.67
15	30	9.56	-0.39
20	37	13.33	-0.24
25	41	15.71	-0.16
50	47	18.44	-0.07

As stated in Table 7.2, a throttle valve opening sweep was conducted at a fixed engine speed of 2000 RPM. The Armfield software automatically applies brake loads based on the engine speed and throttle position, facilitating testing of the engine under various working loads.

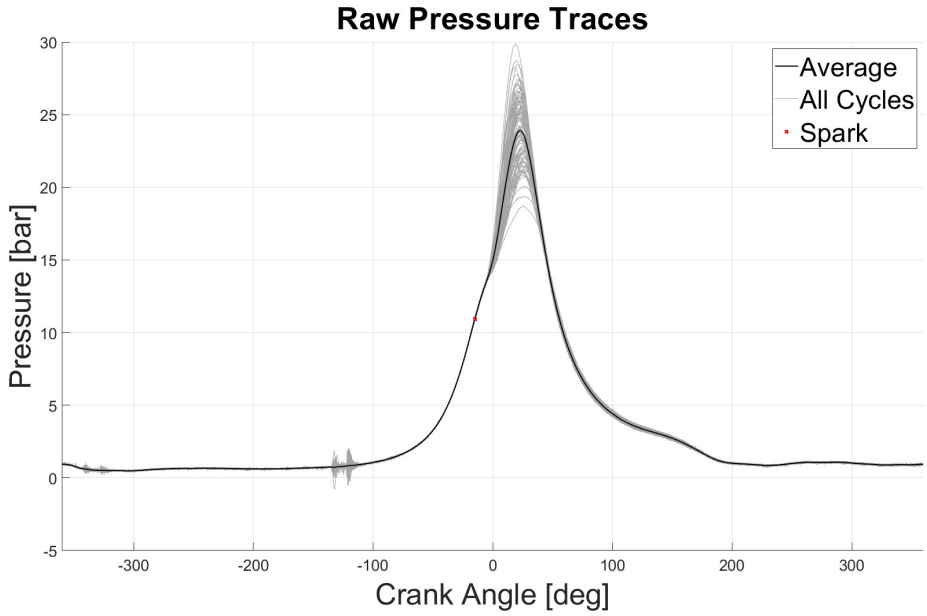
In the first test, the throttle position was set to 9%, and the resulting diagram is displayed in Figure 7.6. Although noise is evident at the end of the intake phase, the signal is sufficiently clear even without filtering.

Figure 7.6: 2000 RPM Throttle position 9%



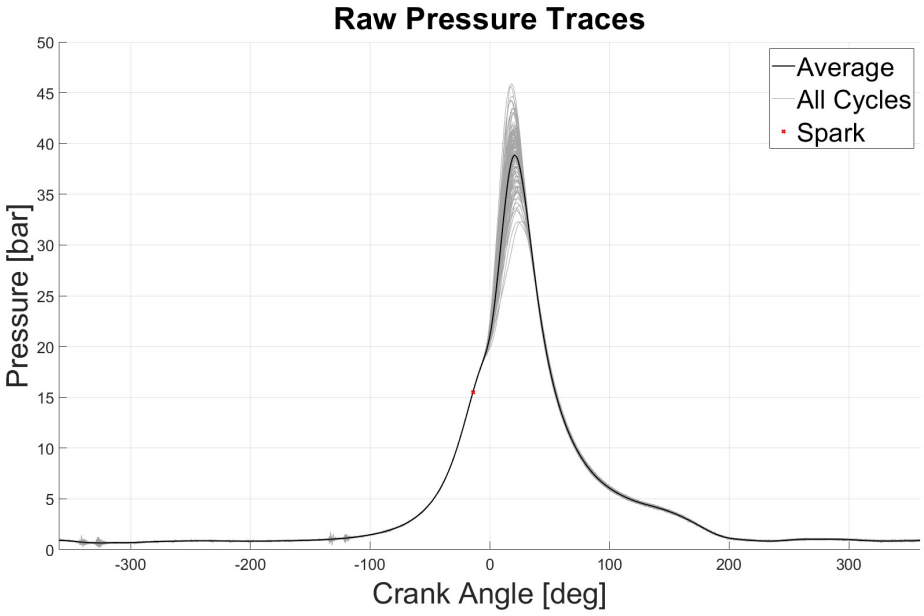
In Figure 7.7 the engine load is increased, the throttle position is set at 15%, and the brake load at 30%. It is possible to see that the pressure peak shifts to the right of the TDC, it can also be noted that the noise during the valve events is less relevant. The cycles oscillate due to the fact that the ECU corrects the combustion parameters as Spark Advance and Injection duration in order to keep the engine in steady-state conditions.

Figure 7.7: 2000 RPM Throttle position 15%



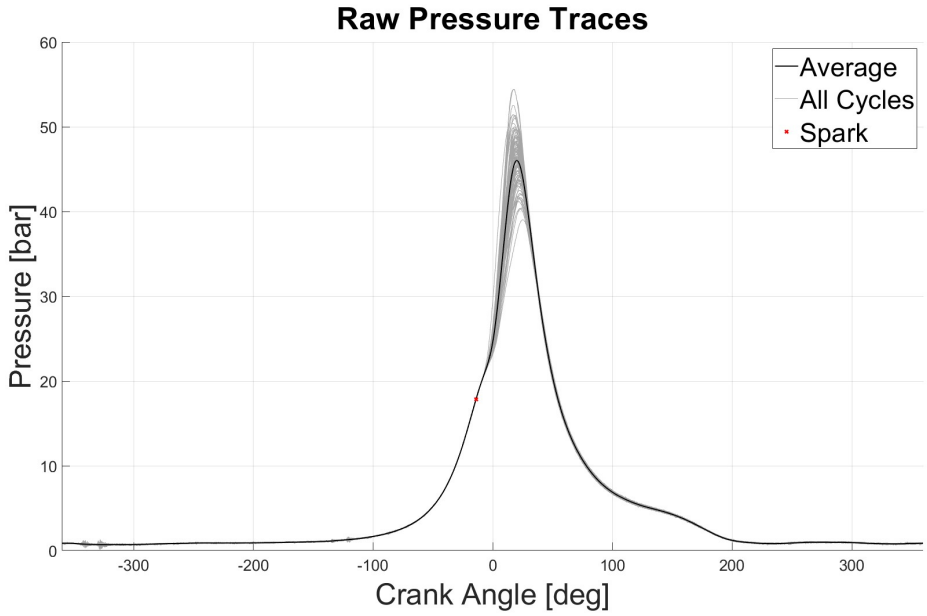
In Figure 7.8 the throttle valve is set at 25% with a brake load at 41% it is possible to see that the combustion starts a bit earlier compared to the low load run. The spark advance is set to 19 ^{o} CA. These curves in the figure are unfiltered, and the noise is not so evident compared to the low-load working conditions.

Figure 7.8: 2000 RPM Throttle position 25%



In the last Figure 7.9 the throttle valve is set at 50% with a brake load at 47%, it is the highest load condition that is tested in this research, and the highest pressure measured is 54 bar. It is possible to notice that as soon as the in-cylinder pressure starts increasing, the noise caused by the valve events decreases. With a throttle valve opening of 9% noise oscillates at a maximum range of 2.8 bar, with a throttle valve opening at 50% the maximum oscillation range is 0.5 bar.

Figure 7.9: 2000 RPM Throttle position 50%



7.3 Consecutive Cycle

Another point that was questioned in this research is how many cycles it is possible to store continuously because of the limitations on the SRAM. This question is reasonable because while the Arduino is recording new data points, it is also saving the data in the queue, and from there it is storing them in the SD Card. Saving measurements instantaneously takes time, for this reason, three engine speeds have been tested: In the tests listed in Table 7.3 every run has stored 1000 engine cycles

Table 7.3: Engine load test at 2000RPM

Engine Speed [RPM]	Throttle position[%]	Power[kW]	MAP[bar]
2800	10	2.43	-0.75
3000	11	3.11	-0.75
3500	12	2.73	-0.77
4000	13	1.93	-0.77

which means that the measurement points for 2000 revolutions are stored in the Arduino. Some of the results of these tests are shown below.

Figure 7.10 displays the pressure traces as a function of the crank angle. As previously mentioned, some noise is present during valve events, as detected by the pressure transducer. However, disregarding this issue (which can be rectified by modifying the Charge amplifier settings), the pressure readings during the combustion phase are relatively clear.

Figure 7.10: 2800 RPM 1000 cycles

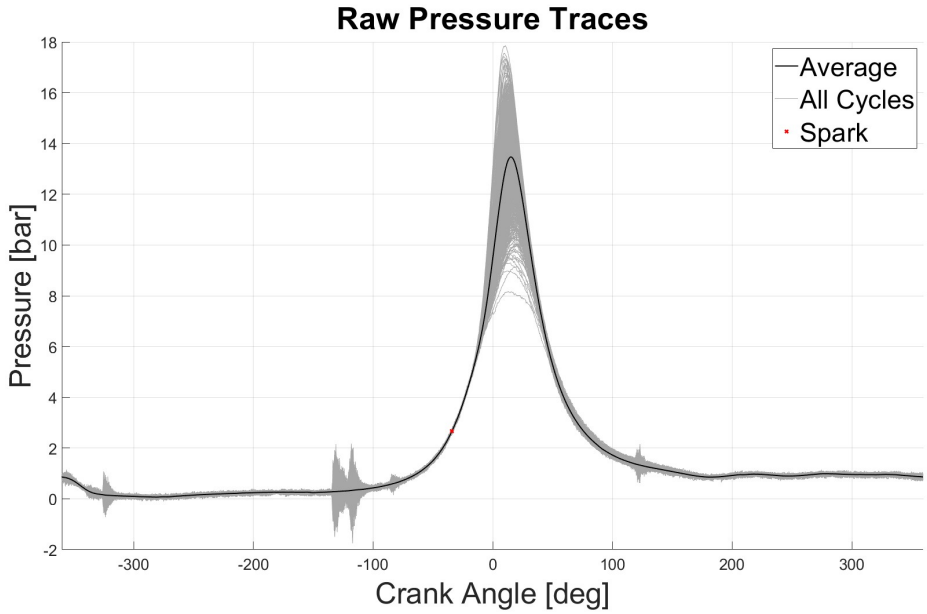
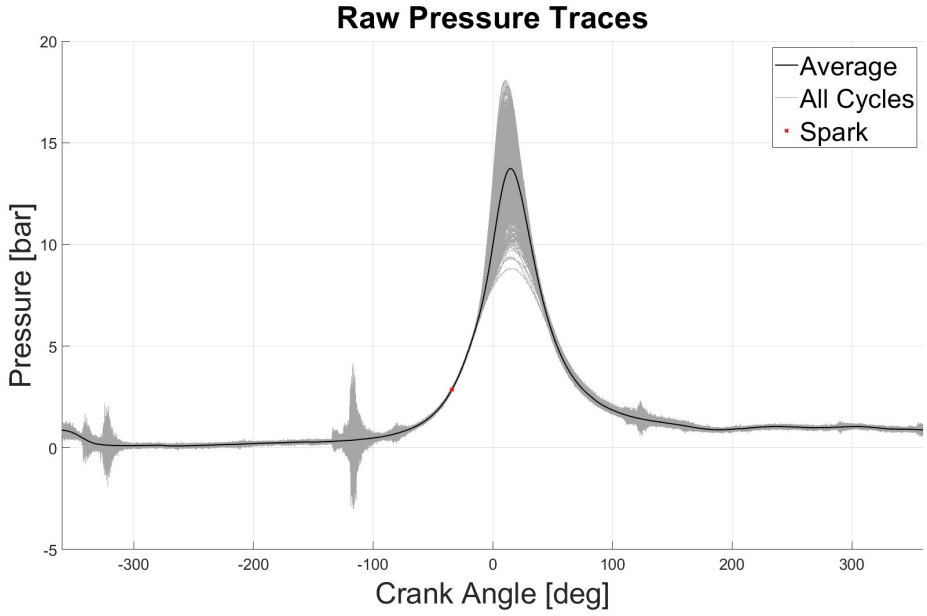


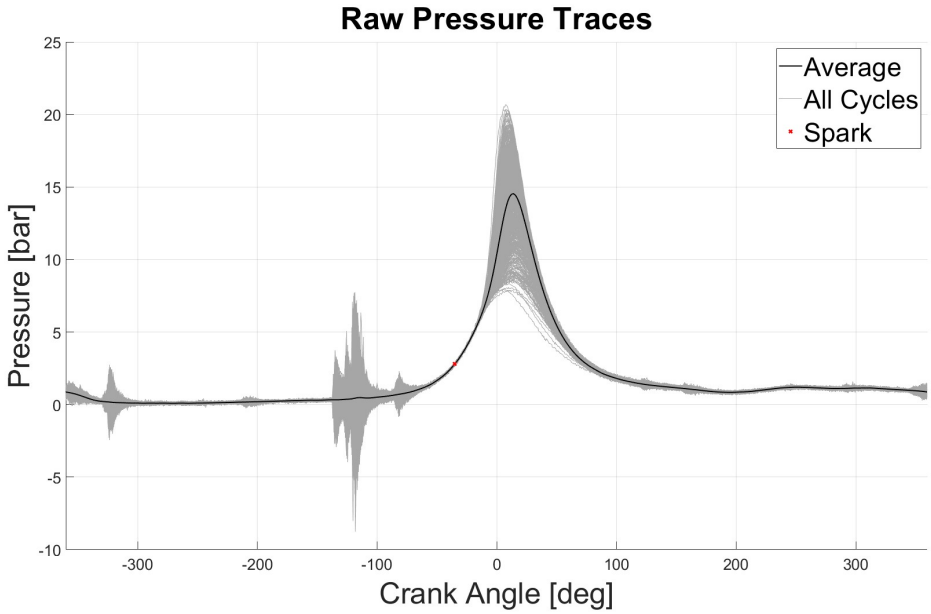
Figure 7.11 shows the results with a run still in low load conditions, it is possible to see that the designed DAQ is still able to record all the engine cycles, with some noise present especially in the intake phase.

Figure 7.11: 3500 RPM 1000 cycles



The last run is held at 4000 RPM and the results are shown in Figure **??**. It is possible to see that the pressure trace in the combustion phase looks good, and over 1000 cycles the variability is high. In order to reduce the amount of noise some filtering can be done through MATLAB, and the results are shown in the next chapter.

Figure 7.12: 4000 RPM 1000 cycles



7.4 Data Filtering

Throughout the previous sections, the pressure signals were filtered for all the runs under consideration. Although only a few examples are presented in this section for comparison, this filtering process was applied consistently throughout. Figure 7.13 displays the filtered trace at 1500 RPM, enabling a comparison with Figure 7.2 which depicts the raw signal. The noise is entirely eliminated in the filtered trace, and, apart from the combustion phase, the pressure measurements during the intake, exhaust, and compression phases are nearly superimposed.

Figure 7.13: 1500 RPM Filtered trace

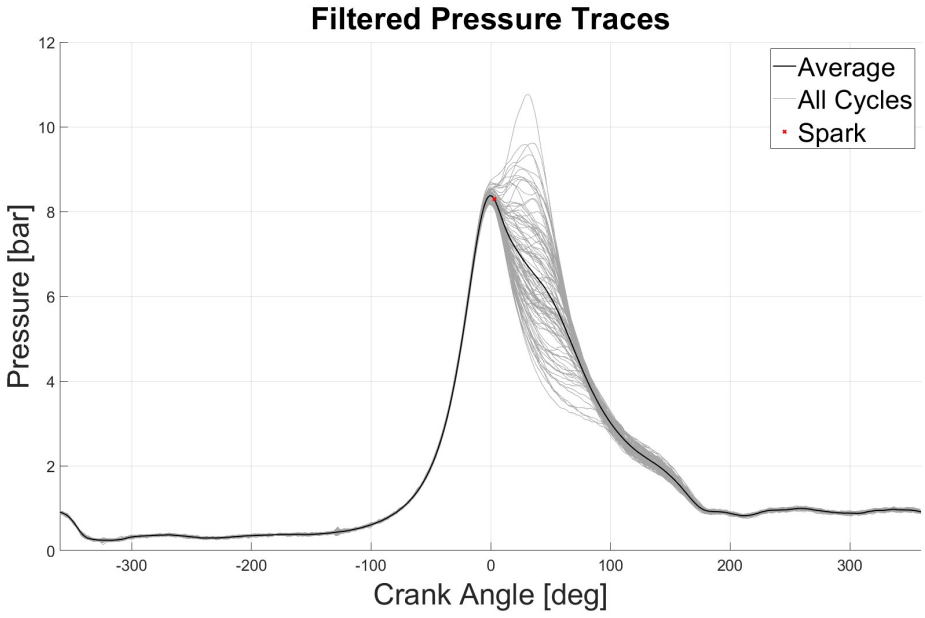


Figure 7.14 displays the pressure measurements taken at 3000 RPM after filtering, as the engine speed increases. In contrast to the raw measurements at 3000 RPM illustrated in Figure 7.4, the valve events are significantly attenuated in the filtered version. This filtering method progressively eliminates the pressure oscillations caused by noise while preserving the overall pressure trend. The last figure related to the Pressure signal vs. Crank Angle degree is Figure 7.15 which is the filtered pressure signal at 4000 RPM. The valve events are evident in this case, but the noise amplitude is not as large as in the raw data. These pressure measurements can be used to study the in-cylinder combustion variability between several cycles, this variability is caused by the combustion process where the ECU controls the parameters in order to have a stable ICE.

Figure 7.14: 3000 RPM Filtered trace

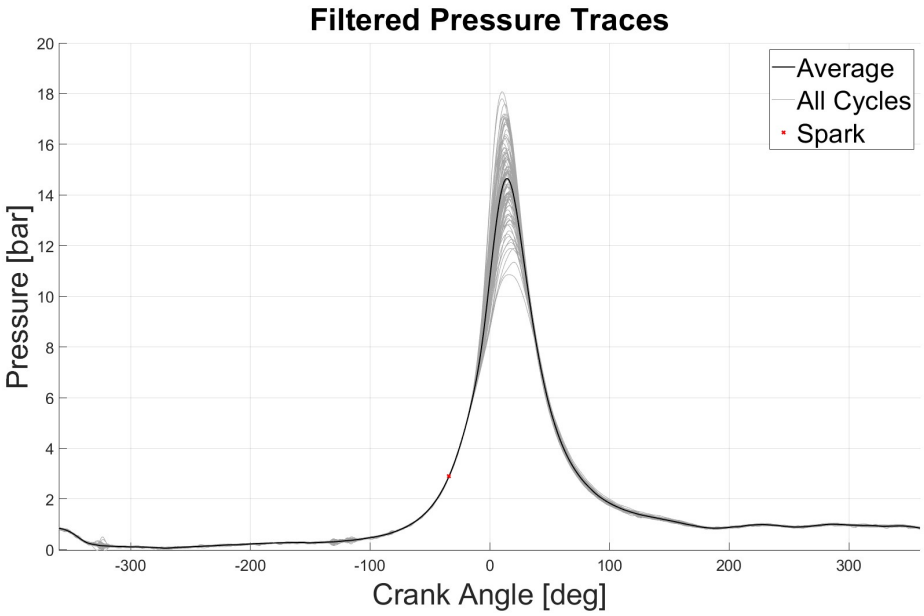
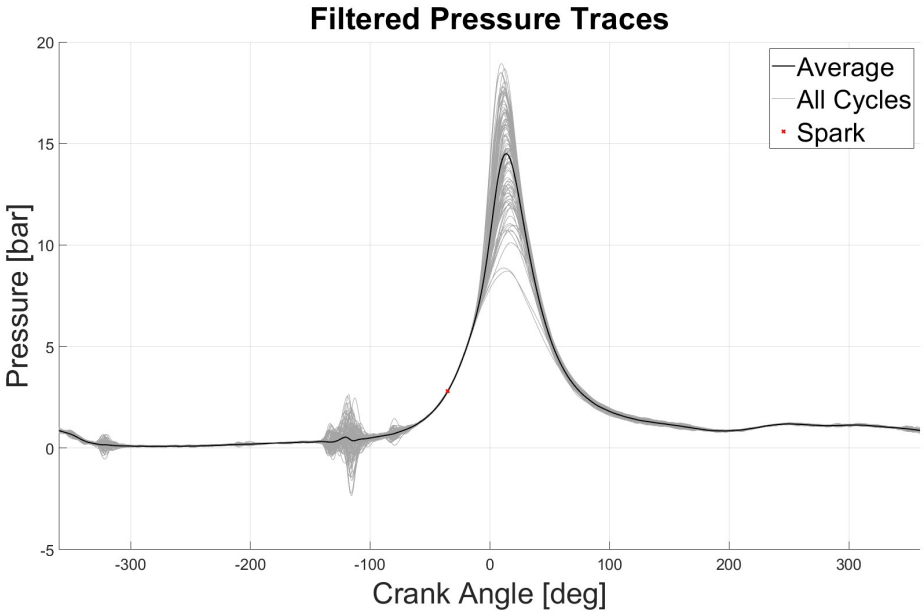


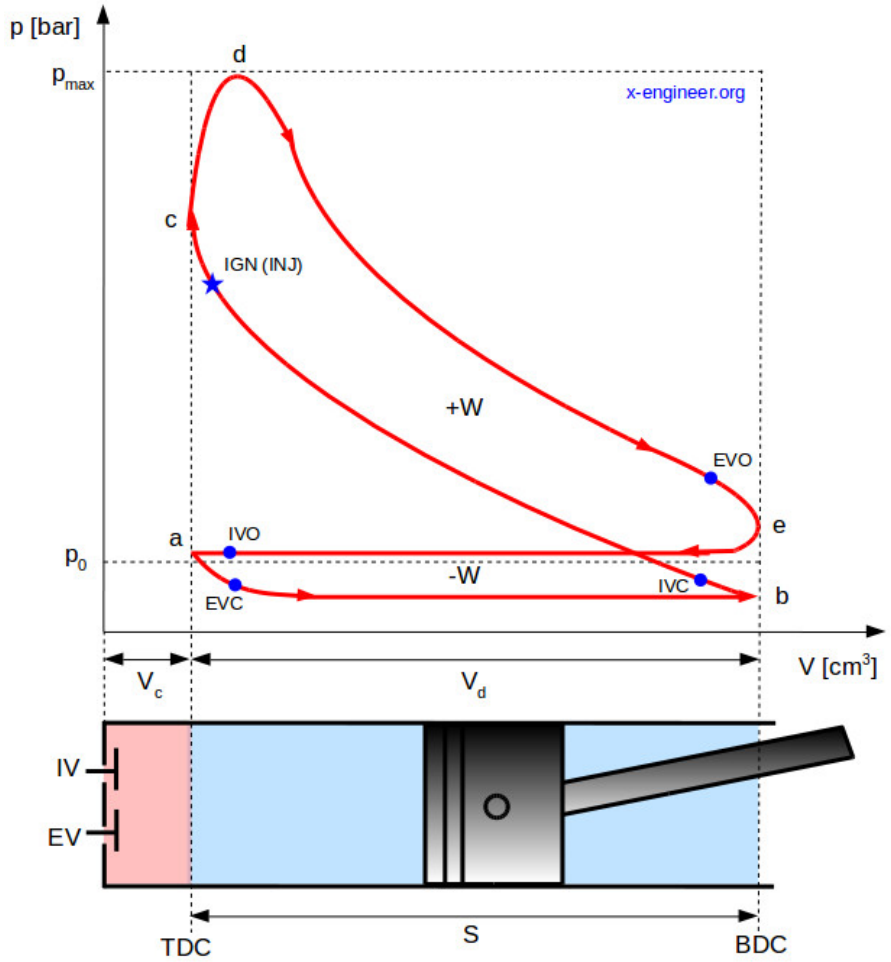
Figure 7.15: 4000 RPM Filtered trace



7.5 P-V diagram

This section presents the P-V diagrams, which are obtained by measuring the pressure in relation to the Crank angle. With this data, it is possible to formulate the equation that describes the cylinder displacement as a function of the Crank Angle degree. This equation portrays the pressure variation inside the cylinder in relation to its volume over a complete engine cycle.

Figure 7.16: PV diagram



Source: x-engineer

In Figure 7.16 the different valve events are defined, they are described below:

- **IVO** Intake Valve Opening, lets the fresh air with fuel mix enter the cylinder to replace the previously burnt gas.

- **EVC** Exhaust Valve Closing, closes the Exhaust valve and allows effectively to fill the cylinder.
- **IVC** Intake Valve Closing closes the intake valve, which makes the compression phase start.
- **IGN** Ignition, is the phase where the combustion process starts due to the spark plug which ignites the fuel present in the mixture.
- **EVO** Exhaust Valve Opening, Opens the exhaust valves to allow the burnt gasses to flow outside the cylinder an replace it with fresh mixture.
- **+W** is the useful work that is released in the combustion process
- **-W** is needed to refill the cylinder with a fresh charge.

In Figure 7.17, the PV diagram captured under idle load conditions is presented, indicating that the engine is solely running to sustain itself without producing any practical output work. The dotted curve represents the cycle-to-cycle variability curves while the semi-transparent area represents the standard deviation of the pressure data.

Figure 7.17: 930 RPM PV diagram

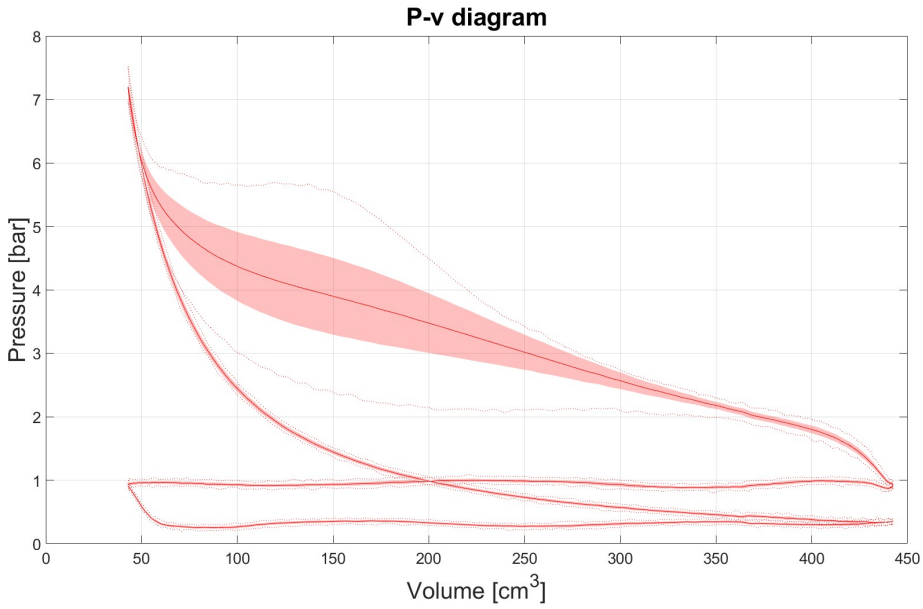
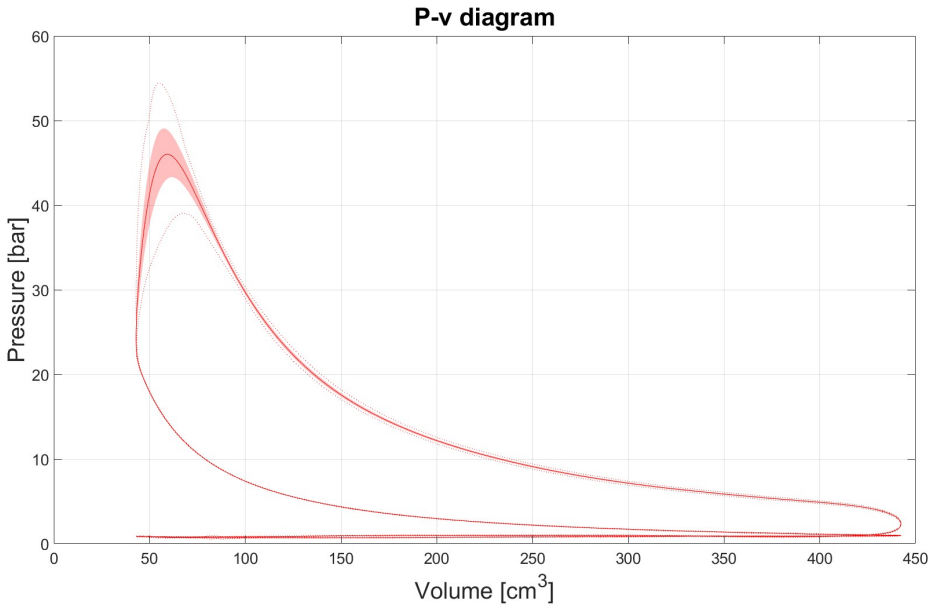


Figure 7.18 depicts the PV diagram obtained at 2000 RPM with a Throttle opening at 50%. A significant difference is apparent when comparing Figure 7.17 and Figure 7.18, notably the considerably larger useful work area in the latter. This discrepancy is primarily attributed to the fact that the output power in the idle working condition is only 0.2 kW, whereas in the latter scenario, it amounts to 18.44 kW.

Figure 7.18: 2000 RPM PV diagram throttle opening at 50%

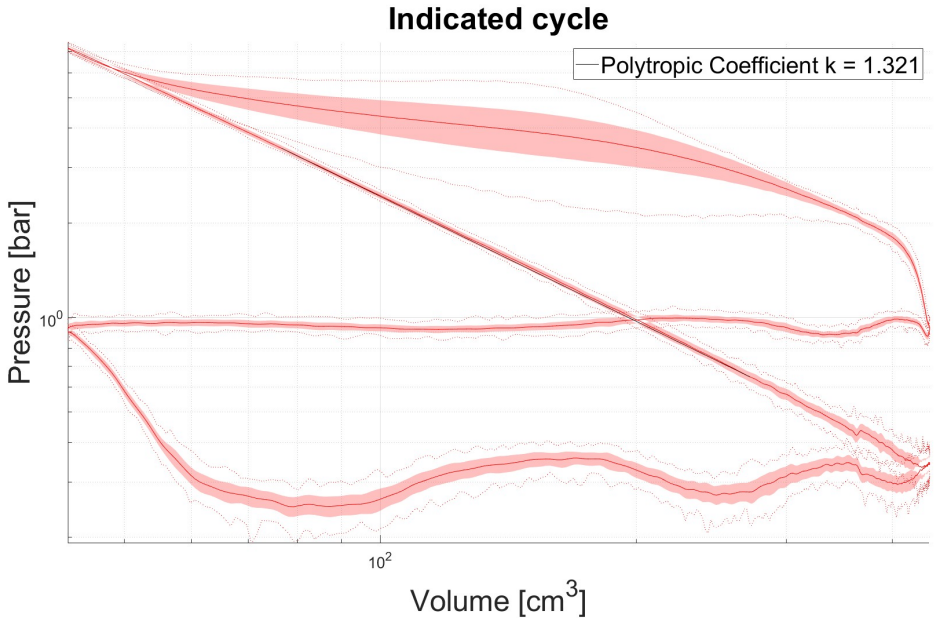


7.5.1 Indicated Cycle

Based on the previously computed data, it is feasible to construct a logarithmic graph of the PV diagram. This enables us to visualize the compression phase, which should exhibit linearity on the logarithmic axis.

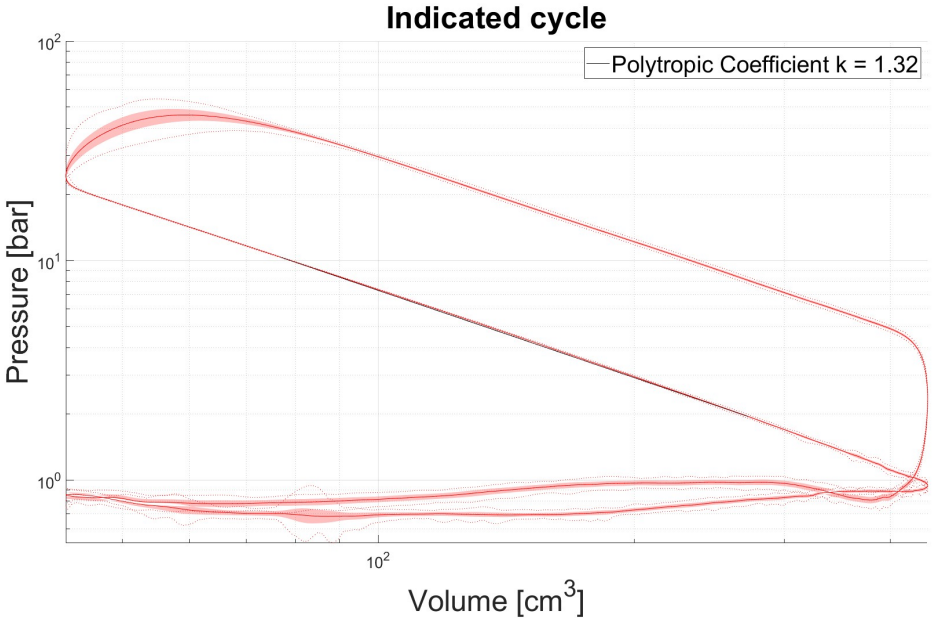
In Figure 7.19 the indicated cycle at idle workload is shown, it is possible to see that the polytropic coefficient, in this case, is 1.321.

Figure 7.19: 930 RPM PV diagram



In Figure 7.20 the high load diagram is shown, in this run the throttle opening is set at 50%, and the polytropic coefficient is 1.32.

Figure 7.20: 2000 RPM Indicated Cycle throttle opening at 50%



It is important to note that a certain degree of discontinuity exists between the measured pressure values with respect to volume and the theoretical trend line. This disparity arises due to the Conrod length of the Armfield testbench, which is not specified in the datasheet. To obtain a Conrod length that aligns with this testbench, extensive research has been conducted on similar engine configurations to achieve the maximum possible accuracy.

Chapter 8

Future development

This chapter discusses the future of the research, building upon the promising results obtained through the Measurement Chain discussed earlier. The aim is to explore modifications that can make the measurements more affordable for Formula SAE teams, hobbyists, and engine builders. Some modifications which can decrease the cost of the setup can be:

- Change the Pressure transducer with a cheaper measurement system;
- Change the Encoder with a less expensive one.

These modifications are discussed in the Sections below.

8.1 Alternative pressure measurement systems

The most straightforward approach seems the application of a piezo-electric washer as a replacement of the original part equipping the spark plug. The main issue affecting the accuracy of cylinder pressure measurement using the piezoelectric spark plug washer is the effect of temperature variations both on the force transmitted by the thread to the washer and piezoelectricity properties. Most of the solutions for the estimation of cylinder pressure is based on indirect approaches. This choice is twofold:

- there is no need for direct access to the combustion chamber, which prevents complex machining operation;
- the sensor doesn't need to operate in a high pressure and temperature environment, making it possible to use cheaper materials and constructive solutions.

These indirect pressure measurements systems can be useful to diminish the price of installing a Pressure transducer, this system shows strong similarities in the

compression stroke with the Piezoelectric Pressure transducer, but the signal diverges during the expansion stroke with systematically higher pressure readings compared to the reference sensor. The possibility of using the washer signal to limit knock intensity is possible using this pressure measurement device[29]. The estimated cost for a piezoelectric washer in mass production is 10 to 100 times lower than the reference sensor depending on the production volumes[30]. Another in-cylinder pressure measurement system can be realized using Optical Fiber, this system has a lower cost compared to the Piezoelectric pressure transducer. Both systems have different responses under thermal shock, the Piezoelectric pressure transducer underestimates the in-cylinder pressure while the optic fiber pressure transducer reacts by either underestimating the pressure measurement or overestimating it.

8.2 Encoder

The encoder used in this research is described in 3.2.2. To decrease the price for the pressure measurement setup, it is possible to choose an encoder that has at least one Absolute output, and one Incremental Output. The absolute output is needed in order to know where the TDC is with respect to the crankshaft rotation.

8.3 Other improvements

One potential improvement for the future could be the development of a real-time streaming system that would transmit pressure measurements from the Arduino directly to a computer. This would enable live monitoring of the in-cylinder pressure, providing more immediate and accurate data. Additionally, efforts could be made to create a more compact design for the system, allowing for quicker assembly, setup, and disassembly.

Chapter 9

Conclusions

The purpose of this study is to develop a Low-Cost Open-Source Data Acquisition for High-Speed Cylinder Pressure Measurement using Arduino. The project is implemented in the Energy Laboratory of Oakland University and then tested on the Armfield CM11 MKII Gasoline Testbench. In Chapter 7 it is possible to find the results of the engine runs on which the Arduino DUE DAQ has been tested. This thesis shows the validation of the DAQ with the requirements described in Chapter 2. It is possible to affirm that the designed Data Acquisition System is capable of:

- Sampling pressure data from Idle speed up to 4000 RPM. This engine speed is set as the target speed, the DAQ is not tested at higher engine speeds due to engine limitations;
- Storing up to 1000 continuous engine cycles at 4000 RPM, it is possible to modify the number of consecutive cycles stored directly from the Arduino IDE software and then upload the new code on the Arduino DUE. The parameter that needs to be modified to change the number of consecutive cycles that the Arduino stores in the SD Card can be found in Section 5.2.1 Line 5
- Storing several data measurements to allow several engine conditions to be tested without removing any data from the Arduino DUE SD Card. Whenever it is necessary, it is possible to remove easily the storage unit from the Arduino due to save the recorded data on a Computer which then allows the post-process of the binary files in pressure measures;
- Reading pressure data coming from the Pressure transducer, the Analog input pins of the Arduino DUE are able to read the voltage output of the Charge Amplifier from 0 to 3.3V, for this reason, the input voltages in the Arduino need to stay within this range.

Overall the results show that the requirements are met. It is evident that the pressure measurements exhibit a high degree of accuracy, although some minor noise is present due to the valve events that occur during high engine speeds and low load conditions. To conclude, it can be confidently affirmed that the Data Acquisition system is not only cost-effective but also holds potential value for Formula Student teams, hobbyists, and engine builders alike. Furthermore, it is noteworthy that the limits of this DAQ system were not fully reached during the course of this research, indicating that further experimentation can be conducted to fully explore the capabilities and limitations of the Arduino DUE-based DAQ system. The present study provides a concise overview of the procedures necessary to install an encoder on the Armfield Testbench. In addition, the electrical and mechanical steps required to connect the encoder to the engine crankshaft and synchronize the pressure readings with the quadrature signal of the encoder on our DAQ are described. The methodology used to calibrate the pressure transducer using a Deadweight tester is also presented in detail to ensure that precise pressure measurements are obtained during the testing phase. In the latter portion of this document, the code for Arduino DUE is presented along with a detailed explanation of the utilized functions. Regarding post-processing, the primary MATLAB functions are explicated, and subsequently, the validation of this data acquisition system (DAQ) is discussed.

Bibliography

[1] *Instruction Manual CM11-MK II*. Armfield Limited. URL: https://armfield.co.uk/wp-content/uploads/2020/05/CM11MKII_Datasheet_v1d_Web.pdf (cit. on p. 2).

[2] *USB1808xDAQ*. Measurment Computing. URL: https://www.mccdaq.com/PDFs/specs/DS-USB-1808-Series.pdf (cit. on p. 3).

[3] *Arduino introduction*. Arduino. URL: https://www.arduino.cc/en/Guide/Introduction (cit. on p. 5).

[4] *Arduino DUE*. Arduino. URL: https://store.arduino.cc/products/arduino-due (cit. on p. 5).

[5] *Pressure Transducer and Charge amplifier Catalogue*. Kistler. URL: https://kistler.cdn.celum.cloud/SAPCommerce_Download_original/960-695e.pdf (cit. on pp. 10, 20).

[6] *Ruland controlflex coupling bundle*. Ruland. URL: https://www.ruland.com/cprs12-5-a-cpfrg12-19-at-cprs12-4-a.html (cit. on p. 12).

[7] *H25 | INCREMENTAL OPTICAL ENCODER*. Sensata Technologies. URL: https://www.sensata.com/sites/default/files/media/documents/2018-05-17/ourproducts_h25_incremental_optical_datasheet.pdf (cit. on p. 17).

[8] *Installation and Maintenance Manual for the ASHCROFT® Type 1305D Deadweight Tester and Type 1327D Portable Pump*. Ashcroft. URL: https://www.instrumart.com/assets/1305D-1327D-manual.pdf (cit. on p. 22).

[9] *Gasoline Port Fuel Injector*. Bosch. URL: https://www.bosch-mobility-solutions.com/en/solutions/powertrain/gasoline/gasoline-port-fuel-injection/ (cit. on p. 25).

[10] *Arduino IDE 2.0*. Arduino. URL: https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2 (cit. on p. 29).

[11] *setup()*. Arduino. URL: https://www.arduino.cc/reference/en/language/structure/sketch/setup/ (cit. on p. 38).

[12] *loop()*. Arduino. URL: https://www.arduino.cc/reference/en/language/structure/sketch/loop/ (cit. on p. 38).

[13] *#include*. Arduino. URL: https://www.arduino.cc/reference/en/language/structure/further-syntax/include/ (cit. on p. 38).

[14] *#define*. Arduino. URL: https://www.arduino.cc/reference/en/language/structure/further-syntax/define/ (cit. on p. 38).

[15] *Unsigned int*. Arduino. URL: https://www.arduino.cc/reference/en/language/variables/data-types/unsignedint/ (cit. on p. 39).

[16] *Int*. Arduino. URL: https://www.arduino.cc/reference/en/language/variables/data-types/int/ (cit. on p. 39).

[17] *Char*. Arduino. URL: https://www.arduino.cc/reference/en/language/variables/data-types/char/ (cit. on p. 39).

[18] *Bool*. Arduino. URL: https://www.arduino.cc/reference/en/language/variables/data-types/bool/ (cit. on p. 39).

[19] *SPI Clock Divider*. Arduino. URL: https://www.arduino.cc/reference/en/language/functions/communication/spi/setclockdivider/ (cit. on p. 39).

[20] *SerialUSB begin*. Arduino. URL: https://www.arduino.cc/reference/en/language/functions/communication/serial/begin/ (cit. on p. 39).

[21] *SerialUSB print*. Arduino. URL: https://www.arduino.cc/reference/en/language/functions/communication/serial/println/ (cit. on p. 39).

[22] *Reset your board*. Arduino. URL: https://support.arduino.cc/hc/en-us/articles/5779192727068-Reset-your-board (cit. on p. 40).

[23] *Reset your board*. Arduino. URL: https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/ (cit. on p. 40).

[24] *SAM3X/SAM3A Series Datasheet*. MICROCHIP. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf (cit. on p. 42).

[25] *fopen*. MATLAB. URL: https://www.mathworks.com/help/matlab/ref/fopen.html (cit. on p. 47).

[26] Andrew L. Randolph. «Methods of Processing Cylinder-Pressure Transducer Signals to Maximize Data Accuracy». In: *SAE Transactions* 99 (1990), pp. 191–200. ISSN: 0096736X, 25771531. URL: http://www.jstor.org/stable/44553970 (visited on 03/15/2023) (cit. on p. 48).

[27] F Payri, P Olmeda, C Guardiola, and J Martın. «Adaptive determination of cut-off frequencies for filtering the in-cylinder pressure in diesel engines combustion analysis». In: *Applied Thermal Engineering* 31.14-15 (2011), pp. 2869–2876 (cit. on p. 49).

[28] Charles A. Amann. «Cylinder-Pressure Measurement and Its Use in Engine Research». In: *SAE Transactions* 94 (1985), pp. 418–435. ISSN: 0096736X. URL: http://www.jstor.org/stable/44467432 (visited on 03/16/2023) (cit. on p. 50).

[29] Enrico Corti, Marco Abbondanza, Fabrizio Ponti, and Lorenzo Raggini. «The Use of Piezoelectric Washers for Feedback Combustion Control». In: *SAE International Journal of Advances and Current Practices in Mobility* 2.2020-01-1146 (2020), pp. 2217–2228 (cit. on p. 74).

[30] Enrico Corti, Lorenzo Raggini, Alessandro Rossi, Alessandro Brusa, Luca Solieri, Daire Corrigan, Nicola Silvestri, and Matteo Cucchi. «Application of Low-Cost Transducers for Indirect In-Cylinder Pressure Measurements». In: *SAE International Journal of Engines* 16.03-16-02-0013 (2022) (cit. on p. 74).

[31] *FIFO Computing*. Wikipedia. URL: https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics).

[32] *The pressure-volume (pV) diagram and how work is produced in an ICE*. x-engineering. URL: https://x-engineer.org/pressure-volume-pv-diagram/.

[33] *Voltage Divider*. Wikipedia. URL: https://en.wikipedia.org/wiki/Voltage_divider.