University of Southern Queensland

Faculty of Engineering and Surveying

Early Detection and Prevention of Catastrophic Failures of Industrial Engines

A Dissertation submitted by

Mr Mikhail Dashchinskiy

in fulfillment of the requirements of

ENG4111 and ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Honours) (Mechanical)

Submitted October, 2017

Abstract

Nowadays, there is a tendency to operate some industrial machinery remotely, such as generator sets, gas pumps, and emergency equipment. It is often impossible to directly interfere with the operation of the equipment in the case of unexpected malfunctions. Premature and abrupt failure of the crankshaft bearings in an industrial engine can consequently lead to catastrophic damage. This project designed and tested an inexpensive system that would detect failure of the crankshaft bearings and provide a quick response.

Based on the extensive literature review and the author's experience, it was decided that the proposed device comprised oil debris monitoring and vibration acquisition processing systems. The rise in pressure across the filtering element was set to indicate the presence of bearing particles in the engine oil, which would trigger a warning. The parameters of the vibration signal, such as spectrogram, Root Mean Square (RMS), and Crest Factor, were selected to monitor changes to the vibration profile associated with the fault conditions. The Matlab Graphical User Interface (GUI) performed the signal processing and displayed all necessary outputs. It also allowed the data recording and data replay options.

The system was tested on a six-cylinder petrol engine with artificially induced failure of one of the crankshaft bearings. The experiments failed to instigate rapid and catastrophic bearing failure due to no load being applied to the engine. Nevertheless, the partial bearing degradation and debris generation were achieved during the test runs.

The outcomes showed that the spectrogram and the RMS of the signal provided some response to the progressing failure. The spectrogram revealed the change in magnitudes of the main harmonic orders and the RMS value. For instance, the first harmonic order became dominant and the RMS behaved erratically. Normally, the third harmonic is the most prominent, and the RMS is proportional to the engine revolutions per minute (RPM). Moreover, the debris detection system caught some of the particles generated by the failing bearing. The debris blocked the filtering element, which caused the pressure to rise across the mesh. At this stage, the system is in its conceptual form. The full functionality, such as the ability to stop the engine based on the threshold parameters,

has not been programmed. The thresholds are still to be properly determined and confirmed by a series of experiments.

Further work is required to check the validity of the first experiments. Additional experiments will need to be performed on the engine whilst mounted to a load cell. Applying the load will allow simulation of the real working condition of the equipment. Future development of the system will require designing a black box solution based on a microcomputer, such Raspberry Pi or similar.

The project acknowledges that the system might be unable to prevent subsequent damage to the components. The primary purpose of the system is to save other components and reduce the repair cost.

Acknowledgment

Assistance in completion of this project is thankfully acknowledged.

Dr Ray Malpress of USQ for providing the supervision and valuable advices.

Mr Jarred StClair of Cavill Power Products P/L for supplying the test engines and all necessary accessories used for the experiments.

University of Southern Queensland

Faculty of Engineering and Surveying

ENG4111 and ENG4112 Research Project

Limitation of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering and Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering and Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitles “Research Project” is to contribute to the overall education within the student’s chosen degree program. This document, the associated hardware, software, drawings, and any other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Certification

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Mikhail Dashchinskiy

Student Number [redacted]

[redacted]

Signature

03/10/2017

Date

Table of Contents

List of Figures

List of Tables

List of Appendices

List of Acronyms and Abbreviations

ADC	Analog to Digital Converter
COM	Communication Port
CPU	Central Processing Unit
AC	Alternating Current
DFT	Discrete Fourier Transformation
FFT	Fast Fourier Transformation
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ISO	International Organization of Standardization
I/0	Input/Output
I2C	Inter Integrated Circuit
IDE	Integrated Development Environment
MEMS	Micro Electro Mechanical System
PC	Personal Computer
PTP	Peak to Peak
RPM	Revolutions per Minute
RMS	Root Mean Squared
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

Introduction

The reciprocating and rotating components of internal combustion engines are subject to frictional wear. The most expensive parts of large industrial engines are cylinder blocks and crankshafts. In some cases, the cost of a new cylinder block may exceed AUD100 000. Premature failure of the crankshaft's bearings may cause catastrophic damage to the cylinder block and render the whole engine unsalvageable.

This project examined a system concept that provided online monitoring of a few critical engine parameters to detect the fault conditions. As part of the project, various approaches to engine parameters monitoring were assessed, and the essential parameters were selected based on the literature review. The project aimed to design, build and test a working prototype of the engine condition monitoring device. For the second stage of the project, the prototype was tested in situ on a passenger car engine, attached to a test bench. The fault condition was introduced into the engine, and performance of the system was evaluated.

Chapter 1 Idea Initiation and Development

1.1 Overview

The aim of this project is to design a system that monitors an engine's vibration, as well as contamination of the lubrication oil. The conceptual design states that the system should be able to detect the event of a crankshaft bearings failure and provide a quick response based on feedback from the accelerometer and oil pressure sensors.

The scope of the project consists of four primary objectives. The first goal is to search and analyse contemporary studies regarding engine conditions monitoring. The second goal is to develop and build a working prototype of a real-time oil condition monitoring system to detect large wear particles generated by abnormal wear of sliding surfaces. The third goal is to build a test rig based on a passenger car engine (6-cylinder inline engine).

The final goal of the project is to evaluate whether the system would satisfy expected performance criteria. In order to achieve this aim, a series of experiments were conducted with the installed prototype. The fault condition was artificially introduced, and a crankshaft damage journal was evaluated after the tests.

1.2 Project Motivation

The current state of worldwide globalisation and the global economy imposes a new set of rules on companies whose main assets consist of machinery. Global competition requires businesses to constantly search for ways to reduce expenses associated with repair and maintenance of equipment. It is inevitable that some components of large assemblies will have premature failures. Moreover, the failure of one component, which itself can be inexpensive, can lead to a chain reaction of consequent

malfunctions, resulting in the costly repair or total write-off of the whole assembly. As such, a recent policy of the main players in the mining and power generation industries is to invest in preventative monitoring and maintenance.

The idea of developing the proposed system arose from the observation of mechanical faults in industrial engines. It was noted that some of the damage was the direct consequence of surface failures caused by friction where, in many cases, the expensive components could have been salvaged.

An initial literature review revealed that few real-time engine condition monitoring systems exist, or are being tested; however, the majority of those systems are focused on maintenance assistance and are also relatively expensive.

Many years of experience have proved that unexpected failures can occur after scheduled maintenance, repairs or overhauls. The literature research identified a knowledge gap in the development of a system that would be easy to implement and which would allow monitoring of the key parameters, sending a warning to the operator or to the engine control module that something is amiss.

Although there will be potential for improvements to the designed system, it will possibly have a market value. It could be used during the dynamometer tests or within the first few hours following a repair.

1.3 Implication

According to Cubillo, et al. (2016), the main reason for premature bearing failure is dirt (45.4%), followed by incorrect assembly (12.8%) and misalignment (12.6%), which can also be considered as improper assembly. Together they contribute to 70.8% of all failures. Based on field experience, it is accepted that dirt is introduced during scheduled repairs and maintenance.

Major causes of premature bearing failure	
Dirt	45.4%
Incorrect assembly	12.8%
Misalignment	12.6%
Insufficient lubrication	11.4%
Overloading	8.1%
Corrosion	3.7%
Improper journal finish	3.2%
Other	2.8%

Figure 1-1 - Major causes of premature bearing failure (Source: Cubillo et al. 2016).

It is widely understood that most premature failures happen within the first few hours after maintenance, repairs or rebuilds. As such, it is paramount to carefully observe all vital engine parameters during those critical hours.

The management systems installed on contemporary engines provide sufficient monitoring of many essential parameters, and are capable of adjusting the engine output based on the readings of pressures, temperatures and load factor. The seizure of a crankshaft bearing, however, is not necessarily followed by low oil pressure, or a noticeable increase in the oil temperature; thus, standard monitoring systems do not have input parameters to detect such events. As a result, the engine continues to run until the friction welding creates a rigid connection between a connection rod (conrod) and the crankshaft. Such conditions subsequently lead to the destruction of the conrod and may damage other components of the engine, such as the cylinder block. Clearly, there is significant need for a system that can detect failure initiation and raise an alarm or stop the machinery automatically.

Several systems that approach this problem in different ways are available on the market; however, the majority of such systems focus on long-term wear and aim to detect slight variations in measured parameters. Moreover, they are relatively expensive and available only for limited applications.

1.4 Research Objectives

Based on the information review (see Chapter 3), the proposed system can be subdivided into vibration acquisition and processing and the oil contamination monitoring systems. It was decided that the system would utilise two microcontrollers to monitor specified parameters. One microcontroller is responsible for reading the data of the engine's RPM and the debris monitoring sensor. The second microcontroller acts as the master controller. It reads the vibration signal, receives data from the slave controller, and communicates with Matlab. A Micro-Electro-Mechanical-System (MEMS) accelerometer (see Chapter 3) and an industrial pressure sensor will be utilised for collecting the data.

At first, all necessary hardware was identified based on specifications such as compatibility with the broad range of peripheral equipment, the accuracy of the analog-to-digital converter (ADC), as well as programming aids such as user-friendly Integrated Development Environment (IDE) and the ability to communicate with Matlab.

- Evaluate the signal processing techniques used in vibration analysis of running machinery.
- Based on experiments, choose which of the features extracted from the vibration signal most accurately reflect the changing conditions of the test engine.

The next objective was to write a suitable Matlab GUI, since it was necessary to automate the implementation of various filtering and processing algorithms. Moreover, it was required to replay and reprocess already recorded data to evaluate various parameters regarding their validity for the engine condition monitoring.

As such, it was intended that the GUI would provide an easy platform for the launching of main subroutines used during the experiments and the post-processing of the data. The key requirements for the GUI are:

- Establish the communication between Matlab and the microcontrollers.
- To provide easy access to subroutines for data logging and data processing.
- To provide visual outputs by means of real-time graphs.
- To possess abilities to replay previously recorded data.

Furthermore, the Matlab program should perform the decision-making process following the established thresholds of chosen parameters; hence, another research objective was to establish the thresholds between normal and abnormal conditions.

The entire project was divided into the following stages:

- Select parameters to monitor based on the literature review (see chapters 2 and 3).Write all necessary Matlab functions and include them in the GUI (see section 4.3).
- Design the system prototype (see section 4.4).
- Write C program for the microcontrollers (see sections 4.4.1- 4.4.3)
- Fabricate the test bench to accommodate the test engine and all necessary subsystems (see section 4.7).
- Conduct initial experiments on the test engine to establish the failure thresholds (see chapters 5 and 6).
- Test the system's response to induced failure conditions based on established thresholds (see chapter 7).
- Analyse outcomes of the experiments (see chapter 8).

The outcomes that the project aimed to achieve were the following:

- Present the working prototype
- Provide valid data of experiments that evaluate the prototype's performance.

- Prove or reject the project's assumption that the system will reduce the damage to the engine components by warning the user of the presence of the fault condition.

1.5 Conclusions

The primary goal of the project was to evaluate the performance of the proposed system. The remit of the project has been extended to develop the Matlab program accompanied by the GUI in order to provide a platform for manipulating the experimental data. The Matlab program can potentially be used in similar experiments as it contains many tools for recording, processing and analysing data obtained from the vibration sensors.

Chapter 2 Literature Review

2.1 Introduction

The following literature review was conducted to evaluate the knowledge gap in the chosen area of research. Initially, the research focused on works relevant to problems identified in the previous chapter, pursuing the following main areas of interest:

1. Failure detection in engines.
2. Condition monitoring of engine oil.
3. Engine vibration monitoring.

Furthermore, since engine vibration monitoring is a complex subject and requires extensive coverage, the literature review incorporates in-depth analysis of this topic.

2.2 Failure detection in engines

Analysis of the literature revealed that several methods of real-time engine condition monitoring have been developed and tested; however, the majority of designs and already-tested solutions concentrate on the progressive degradation of internal components caused by natural wear. Limited information was available on studies aiming to detect rapid and unpredictable degradation of journal bearings of conventional reciprocating engines.

Three main types of monitoring system can be distinguished:

1. Lubrication oil condition monitoring.
2. Vibration analysing systems.

3. Systems that combine both methods.

Little literature can be found regarding the second and the third types. A few sources discovered theoretically considered using parameters such as vibration level in combination with a fluctuation of the torque and the combustion pressure, as the reference parameters to indicate the fault state of an engine (Watzenig, et al., 2013).

2.3 Engine Oil Conditions Monitoring

For the majority of researchers, the area of interest is centred on the development of reliable solutions for real-time oil condition monitoring. The well-known and widely accepted oil testing technique involves manually taking oil samples on a regular basis and performing laboratory analysis. This method has been proven to provide accurate results that could be used for the scheduled maintenance of a customer's equipment (Agoston, et al., 2008). The proposed research topic, however, focused on real-time condition monitoring; hence, the literature review was mainly comprised of studies regarding online systems.

The central principle behind online monitoring is to test parameters of the oil passing through the device at the constant flow rate. It was possible to identify two most common concepts in this field:

- The ferromagnetic particles count method, which is based on a sensing of the electromagnetic field disturbances as wear particles enter a tube, with the induction coil located outside of the tube (Stecki, 1980; Lewicki, et al., 1992; Du, et al., 2010; Dupuis, 2010; Zhu, et al., 2013)
- Monitoring the degradation of the oil by examining properties such as the dielectric constant and transparency (Kumar, et al., 2005; Raadnui and Kleesuwan, 2005; Agoston, et al., 2008).

Some researchers exercised other approaches; however, to some extent, they represent variations of both methods described above.

Despite the extensive work that has been done in this area, Yan, et al. (2013) point out that we have yet to develop an adequate system that can accurately estimate the condition of components based on the oil analysis. There is still considerable uncertainty among researchers as they try to specify the threshold for the size of a wear particle that would signal the presence of a fault condition (Macián, et al., 2003). Zhu, et al. (2015) argued that abnormal wear would generate particles of 20µm to 150µm in size, while Matsumoto, et al. (2016) proposed that particles would need to be larger than 30µm for bearing failure to occur. Vališ, et al. (2015) targeted particular metals such as iron (Fe) and lead (Pb), which are the base constituents of sliding bearings. They examined dependencies between different bearing materials and contaminants released into the oil during normal and abnormal conditions.

2.4 Engine vibration monitoring

Engine vibration monitoring is not as highly regarded as oil condition monitoring due to uncertainties amongst researchers regarding what features of the vibration signal fault conditions. Geng, et al. (2003), Muñoz, et al. (2012) and Peng, et al. (2005) concentrated on acoustic emission and attempted to correlate the impact of excitations in engines occurring during mechanical impacts, such as opening/closing of inlet/outlet valves, and the fuel combustion. Their work is based on applying wavelet filtering (see chapter 3 for detailed review of filter algorithms) to extract impact signatures presented in the signal, and synchronising them with the angular position of the crankshaft. Comparing the energy of the acoustic emission and the magnitude of the vibration with the reference values, it is plausible to indicate abnormal conditions occurring in the engine. Furthermore, Taghizadeh-Alisaraei, et al. (2016) applied the Fast Fourier Transformation (FFT) and wavelet filtering, combined with a mapping of the spectrum of the signal against the crankshaft's angular position. They concluded that abnormal conditions, such as knocking or improper fuel combustion, would induce specific frequencies that are not present in the reference signal.

Broatch, et al. (2008) suggest that measuring the angular acceleration of the engine block reveals fluctuations in the torsion vibration caused by abnormal firing conditions in a cylinder. In their work, they found that, under normal working conditions, the torque pulses are proportional to the angular acceleration of the engine block about the axis of the crankshaft's rotation. As such, the main torque pulses produce a change in angular acceleration that is equal to the firing frequency of the engine, which in turns is a multiple of the rotational frequency. A failure in one of the cylinders, however, causes the firing frequency to become a fraction of the rotational frequency.

Moosavian, et al. (2016) based their work on extracting and comparing statistical features of the vibration signals, such as RMS, Skewness, Kurtosis and Crest Factor. Their experiments demonstrated that the induced fault condition might cause a significant increase in values of outlined parameters.

2.5 Conclusions

From a scan of the literature, it was impossible to identify any relevant source describing a system that could prevent the catastrophic destruction of the main components of an engine. Most of the systems on the market, or those that are currently proposed, mainly concentrate on fine particles oil analysis, whose aim is to provide component condition monitoring over the long term.

The literature review failed to uncover any data regarding experiments testing the proposed systems at critical conditions such as abnormal or rapid bearings degradation. No comparison has been made to demonstrate the performance of such systems when sudden crankshaft bearing failure occurs. A few similar experiments were conducted on gearboxes and test engines (Yibo, et al., 2010; Li, et al., 2012). Despite the tests yielding positive outcomes, they were conducted on the piston/liner group and the valve mechanism where failures of these components usually do not lead to catastrophic damage of engines. Moreover, during the experiments, the test engines were driven by electric motors, providing only partial simulation of the real operational conditions

Chapter 3 Information Review

3.1 Introduction

Several parameters can be monitored to detect a fault condition of a running engine. These can be separated into two classes. The first relates to problems in the engine's thermodynamic cycle. Parameters such as the combustion pressure and the exhaust temperature may indicate faults in intake, exhaust or fuel systems. The second class of parameters is used to determine problems associated with rotating and reciprocating parts of the engine. Engine oil contamination, a loss of power, temperature rise and vibration monitoring can provide real-time assessment criteria for the detection of fault conditions. Since the project’s aim is to detect the failure of the plain bearings, particularly crankshaft bearings, the second class had been considered in detail.

3.2 Wear particles detection

According to Cubillo, et al. (2016) abrasive wear contributes to around 60% of bearings damage.

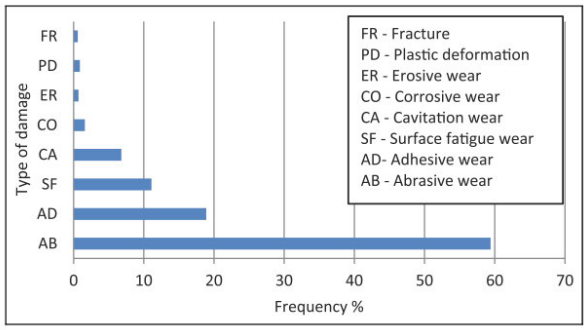


Figure 3-1 - Types of bearing damage and frequency of occurrence (Source : Cubillo et al. 2016).

Wear particles that contaminate the oil are released due to friction between sliding surfaces. To determine the particles of interest, it is necessary to consider the composition of crankshaft bearings.

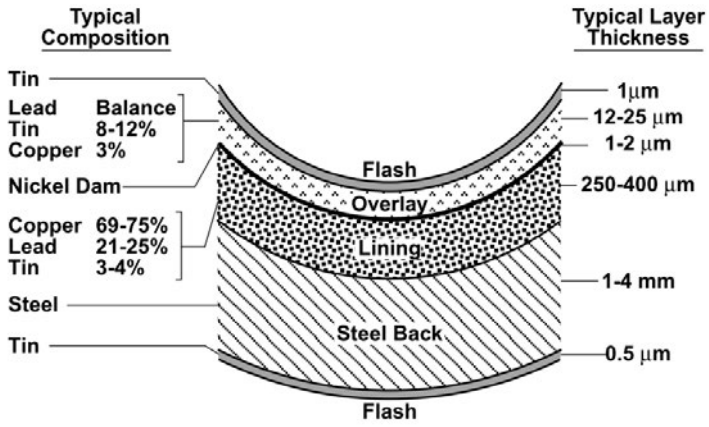


Figure 3-2 - Journal bearings composition (Source : McGeehan & Ryason 1999)

In Figure 3-2, it is shown that the plain crankshaft bearing consist of three primary layers, as well as two flash layers that wear away in the first hours of the first run. During normal operation, the wear interface is established between the crankshaft

journal and the second layer, which is called 'overlay'. Laboratory oil tests are designed to detect the concentration of lead and tin as representatives of the machine condition. The typical particle size ranges between 10–20µm for the average condition (see Figure 3-3) and increases towards 50–100µm in the case of abnormal wear (Du, et al., 2010; Matsumoto, et al., 2016).

In failure mode, the overlay layer rapidly disintegrates, leaving the copper-based lining exposed. The copper lining is soft and porous, apposite to the generation of large flakes of metal. Since the proposed system is aimed at detecting advanced or catastrophic failure, the target size range chosen is 1–10mm.

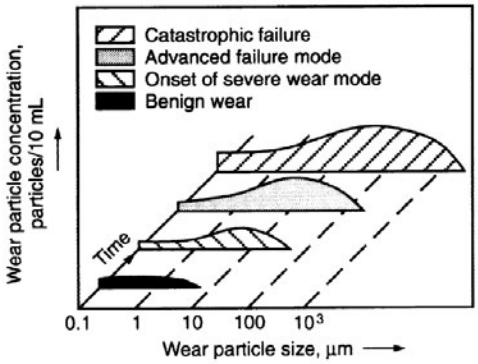


Figure 3-3 - Wear particle size relative to the failure mode (Source : Macián et al. 2003).

3.2.1 Inductive method of the wear particles count

The most widely accepted methods of wear particles detection are the online Ferromagnetic Particles count or the Magnetic Induction count. These kinds of systems are placed in parallel to the main oil galleries in order to continuously monitor the amount and size of wear particles passing through the sensor. The sensing algorithm is based on detecting a disturbance in the magnetic field caused by metal debris. Each disturbance is counted as a wear particle. By counting the number of disturbances per

set time and knowing the flow rate, the system provides an output of the number of particles per control volume.

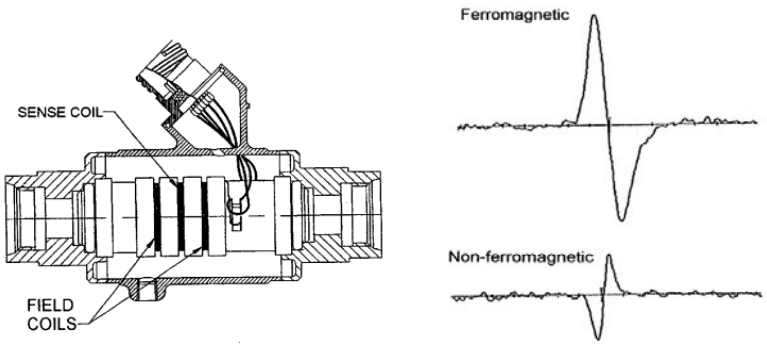


Figure 3-4 - Inductive wear debris sensor and the output signal (Source: Miller & Kitaljevich 2000)

The inductive method requires relatively expensive sensors and a control module; thus, it is generally only used in aircrafts and large marine engines where the good condition of the components is critical. Moreover, the reviewed literature examined the performance of this system for long-term wear detection. The actual behaviour of the sensors under extreme conditions was unclear.

3.2.2 Pressure drop method

Hunt (1995) refers to a relatively simple and reliable method of detecting excessive oil contamination by wear debris.

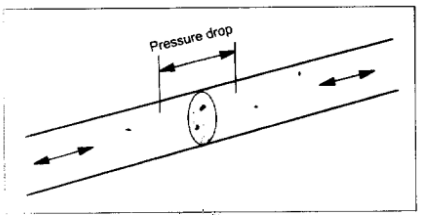


Figure 3-5 - Principal of the pressure drop method (Source :Hunt 1995).

The pressure drop method deploys two pressure sensors placed before and after a filtering element. As the filter becomes blocked by debris, a decrease in pressure across the element is detected and an alarm is triggered. The main limitation of this system is that its sensitivity depends on the mesh size of the filter; however, this disadvantage can be overcome by placing a series of meshes with additional sensors. The advantage of this method is that it does not require expensive equipment and complicated signal processing algorithms. This approach has promising potential in being able to detect catastrophic failure. Moreover, the system can be simplified by reducing it to one pressure sensor located upstream. In this case, blockage of the element will cause the pressure to rise, which can be incorporated as the triggering point for the alarm.

Several points need to be taken into account. Firstly, the system may become blocked by soot or by contaminants introduced during regular maintenance; therefore, the system requires regular inspection and cleaning. An alternative to cleaning could be to create a backflow by reversing the oil pump. Secondly, the mesh restriction should be considered. The effect of increased viscosity of the cold oil may cause issues if equipment operates in a cold climate. The chapter 5 will describe a set of experiments that evaluate the effect of the mesh size on the flow restriction at different oil temperatures.

3.3 Vibration Signal Acquisition and Processing

3.3.1 Definition of Vibration

Vibration can be viewed as a by-product of forces that act within a system. In other words, it is the wasted energy that could be used to produce some work. Vibration, however, is a rather more complicated phenomenon. Excessive vibration can accelerate the fatigue of components, causing their premature destruction. In conventional engines, unwanted vibrations are managed by balancing the rotational masses with counterweights or balancing shafts. The torsion vibrations that accompany the torque pulses can be suppressed using some viscous or rubber dumpers.

It has been found that vibration signatures produced by machinery change as the condition of rotational components deteriorate. As such, by analysing the variance of the vibration signature of a healthy component with current signal, it can be revealed that some abnormality is developing in the component (Moosavian, et al., 2016).

Vibration signatures produced by reciprocating engines are much more complicated than those generated by steady rotating machines, such as electric motors or gearboxes (Geng, et al., 2003). The process of extraction of the useful information from the complex signal requires various techniques that will be discussed further.

Vibration is often examined according to its two main features, which are represented by the time domain and the frequency domain. It is possible to analyse vibration history in the time domain implementing the statistical approach, whereas mathematical methods are used to analyse vibrations in the frequency domain (Serridge and Licht, 1987).

3.3.2 Signal Sampling

i. Sampling Frequency

To avoid the aliasing effect, the sampling frequency must be twice as high as the maximum frequency of the sampled signal. This approach is known as the Nyquist theorem (Misiti, et al., 1996). The processes, such as combustion and the opening and closing of valves, occur approximately nineteen times per revolution for a six-cylinder inline, four-stroke petroleum engine. For the test engine, the maximum RPM during the trial did not exceed 2000–2500rpm; hence, to satisfy the Nyquist theorem, the minimum sampling frequency is:

$$fb = \frac{2500}{60} * 2 * 9 = 750Hz$$

The sampling frequency (Nyquist rate) must be not less than 750Hz. The developed signal acquisition system has the sampling rate of 2kHz, which provides sufficient background for avoiding the aliasing effect (Levis, 2011).

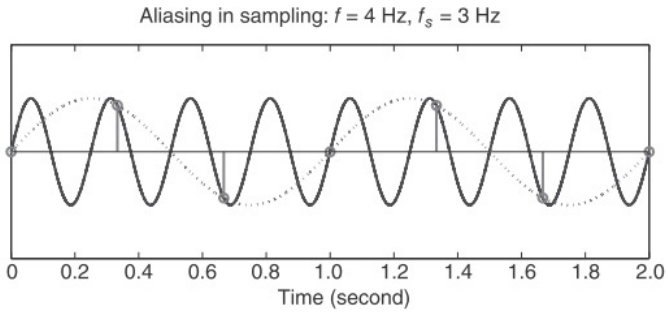


Figure 3-6 - Aliasing effect (Source: Levis 2011)

ii. Discrete sampling vs. RPM referenced sampling

The sampling of the vibration data can be accomplished by either a fixed number of samples per second or a certain sampling rate per revolution. The first method is relatively straightforward to implement; however, the output signal reveals only the general content of the vibration, such as harmonics orders, RMS, Crest Factor and other parameters.

RPM referenced sampling requires an impulse to be generated as the output shaft passes 360 degrees, to synchronise the sampled data with the angular position of the shaft. This approach relies on more complex equipment than for the discrete sampling. The advantage of this method, however, is that it allows the determination of a certain event occurring along the full revolution of the shaft (Sujatha, 2010).

3.3.3 Signal Processing

i. RMS

Peak-to-peak (PTP) magnitude is one of the main features of the vibration signal that indicates the state of a system. The direct consequence of abnormalities occurring inside of a component can be an excessive level of vibration. Comparing the current PTP with the reference magnitude, one can detect when the functionality of the engine is deteriorating; however, PTP provides a poor representation of the signal's state since it does not take into account the time history of the signal. The RMS can be used to overcome this issue, as it represents the average level of the signal, or in other words, the energy content of the signal.

The RMS is a square root of the mean of the sum of samples squared, and can be calculated using the following formula:

$$RMS = \sqrt{\frac{a_1^2 + a_2^2 + a_3^2 + \cdots + a_n^2}{n}}$$

Where:

a =Value of a sample

n =Number of samples

The RMS of the signal measures the vibration energy content. Comparing the current RMS with the reference RMS provides an estimation of the state of the component. Increasing RMS value observed for the same working condition over a fixed period provides a good indication of a problem developing in the system (Sujatha, 2010).

ii. Crest Factor

The Crest Factor represents the ratio of peak value to the RMS. Although the RMS value of a signal is a valuable parameter when the total vibration energy increases, faulty components can generate random peaks of high magnitude that might not affect the RMS value considerably. For this reason, to increase the sensitivity of the RMS method, the ratio of the largest peak to the RMS, also called the Crest Factor, was introduced.

In general, the short and sharp pulses that are generated by a worn bearing may not contain enough energy to affect the RMS readings; however, increasing value of the Crest Factor indicates the origination of a system fault or the presence of erratic vibrations. Another apparent aspect of the Crest Factor is that it is insensitive to the speed and load factor. When an increase in the engine speed causes a rise in the RMS value, the Crest Factor remains the same. This feature makes the Crest Factor a good parameter for a visual representation of the vibration data statistics (Sujatha, 2010).

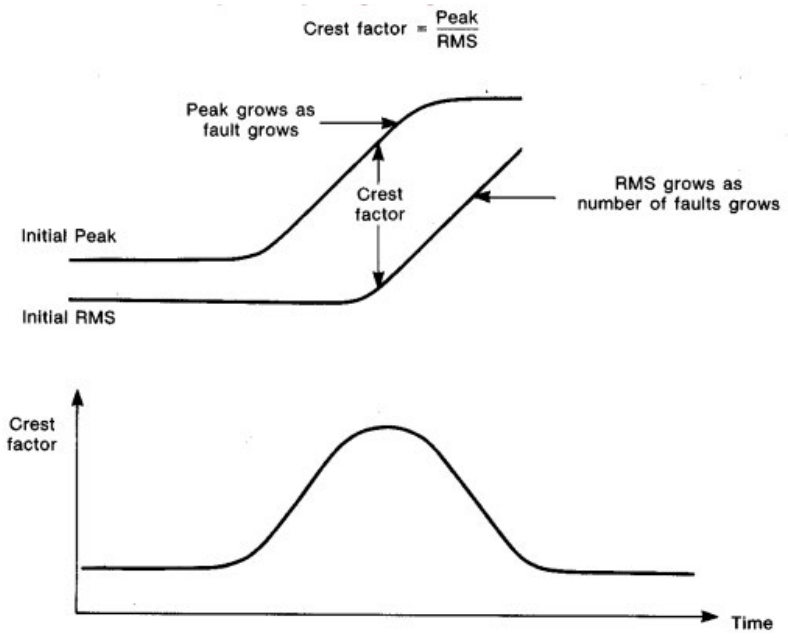


Figure 3-7 - Representation of the relation between the RMS value and the Crest Factor (Source: Unknown)

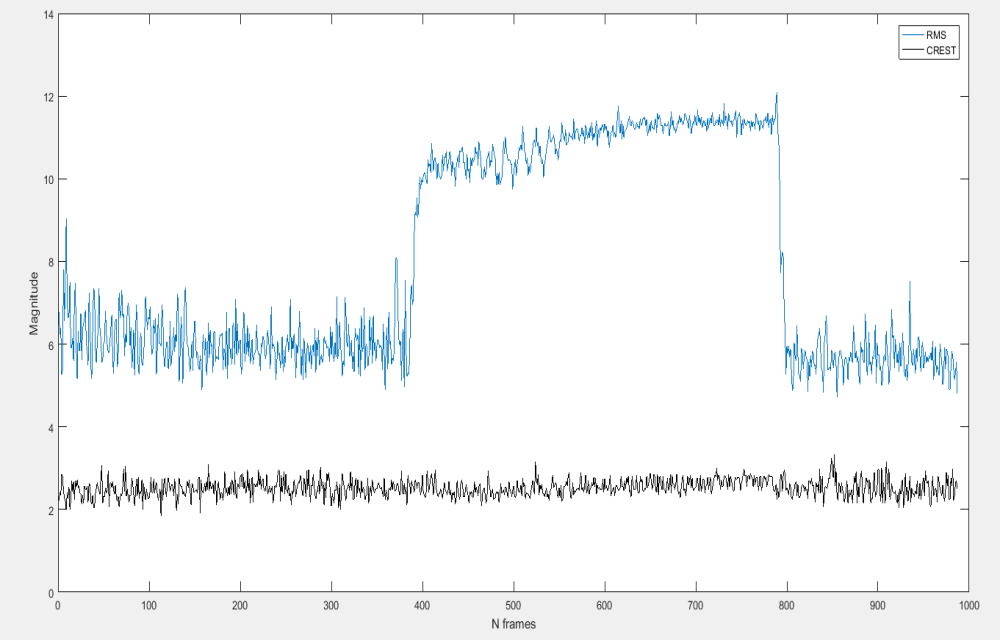


Figure 3-8 - Comparison of the RMS and the Crest factor (Produced with the aid of MATLB 2016A student version).

Figure 3-8 depicts a vibration signal obtained from a running engine. The plot represents a thousand frames of data, where each frame contains five hundred samples. As seen at the beginning of the plot, the RMS (blue curve) corresponds to a certain RPM. At the point of four hundred frames, the RPM doubled, which affected the RMS value, whereas the magnitude of the Crest Factor (black curve) remained unchanged.

iii. Kurtosis

Kurtosis is a statistical term that is calculated as a ratio of the fourth order statistical moment to the second order of the moment (Bruel and Kjaer, 2009).

$$K = \frac{\frac{1}{N}\sum(x_i^4)}{(\frac{1}{N}\sum(x_i^2))^2}$$

Where:

N =Number of samples

X = Value of the sample

Kurtosis provides a similar output to the Crest Factor and is entirely independent of the load factor and rotational speed of the shaft. Kurtosis describes the number of peaks in a signal; hence, the more random peaks present, the higher the Kurtosis number. Sujatha (2010) highlights that Kurtosis is more sensitive to sharp spikes in the signal; however, as the fault progresses, the number of spikes increases, causing the Kurtosis value to return to normal. As such, Kurtosis is only suitable for the early detection of a fault.

In Figure 3-9, the same data of a running engine that was used in Figure 3-8 is shown. Here, the plot illustrates the correlation between the Crest Factor (blue curve) and the Kurtosis values (red curve). It is worth noting that the Gaussian signals (signals with noise normally distributed according the Gaussian probability density function (Sujatha 2010)) have Kurtosis values around three.

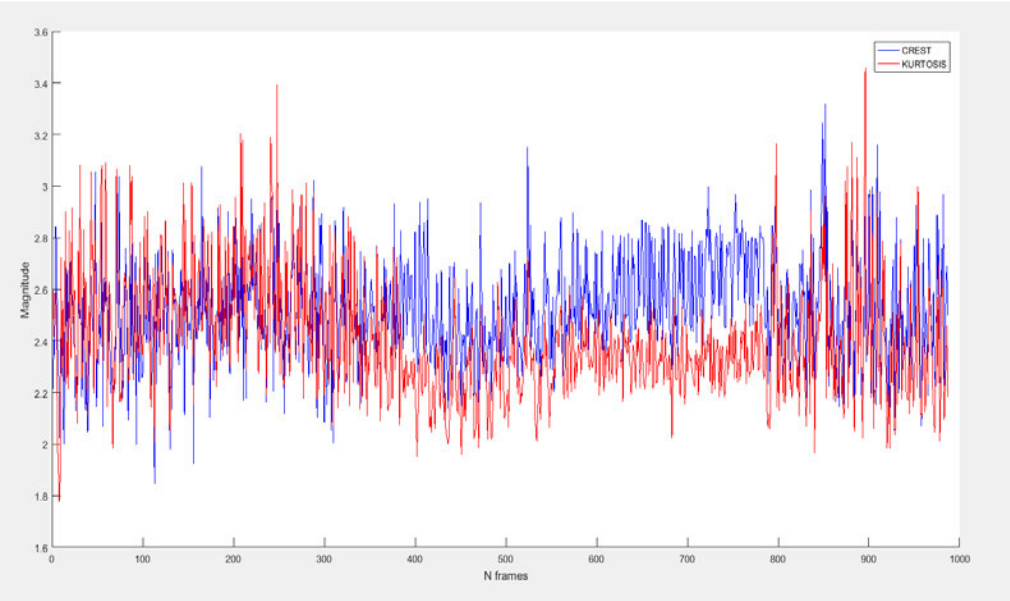


Figure 3-9 - Comparison of the RMS and the Kurtosis (Produced with the aid of MATLB 2016A student version).

iv. The decibel scale

The decibel scale represents a ratio of the measured signal to the reference signal in a logarithmic scale. It can be used to compare the measured decibel level with the level defined by the user. For instance, one can say that the measured signal is 20dB above the base signal.

$$N(db) = 20\log\left(\frac{a}{a(ref)}\right)$$

Where:

a= measured vibration amplitude

a(ref) = reference amplitude. According to International Organization of Standardization (ISO) the value of a(ref) = 10e-6 m/s^2 (ISO 1683, 2015). The same decibel scale can be applied to the RMS value (Serridge and Licht, 1987).

v. Waterfall Analysis

The waterfall analysis is a three-dimensional plot of a vibration signal where the X-axis represents the frequency domain, the Y-axis shows the time domain, and the magnitude is plotted on the Z-axis. Although waterfall analysis is not suitable for real-time signal processing, it provides a good representation of the history of the signal. It is extensively used to determine what happens during transient conditions. It reveals trends in the signal that may arise due to the initiation and progression of fault conditions.

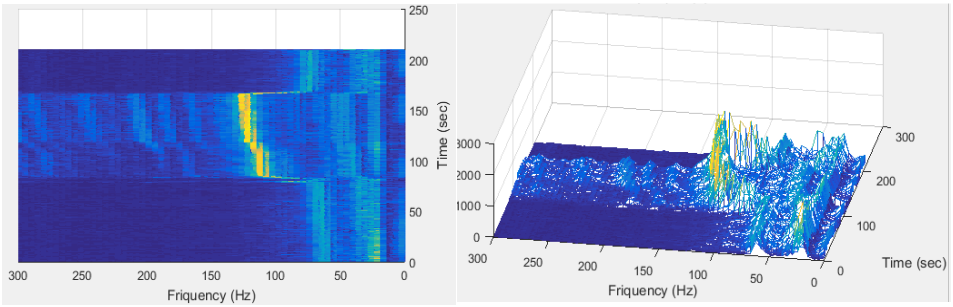


Figure 3-10 - Waterfall plot 2D view (left) and 3D view (right) (Produced with the aid of MATLB 2016A student version).

Figure 3-10 depicts the vibration signature of a six-cylinder four-stroke inline engine plotted as a waterfall. Initially, the engine runs at 1100rpm. The first and third order harmonics can be distinguished as the vertical lines (light blue). They represent a rotational frequency of 18Hz and a firing frequency of 55Hz, respectively. After approximately 100 seconds, the RPM increased to 2200rpm. As shown, the firing

harmonic became more prominent (yellow region) and caused the resonance to be excited.

vi. Fourier Transform Analysis

Discrete Fourier Transformation (DFT) states that every signal consists of many sine waves with various frequencies overlaid on top of one another. DFT allows the signal to be decomposed into its sinusoidal constituents. Afterwards, the DFT results can be plotted as a function of the magnitude and corresponding frequency. The plot is often called the spectrum of the signal. The DFT method provides a high resolution of the spectrum of the signal.

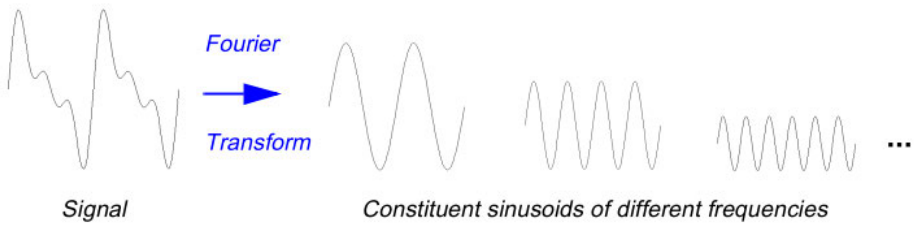


Figure 3-11 - Sinusoidal components of complex signals (Source :Misiti et al. 1996)

vii. The FFT

The high accuracy and resolution of the DFT method come at the cost of computational time. In applications that require real-time signal processing, applying the DFT method puts limitations on the maximum allowed sampling frequency. The introduction of Fast Fourier Transformation (FFT) made it possible to overcome the sampling rate limitations of DFT. The key idea of the FFT method is that the signal is divided into frames (see Figure 3-12).

viii. Framing or Buffering

Signal samples are grouped into an array, or buffer, to perform the FFT. The more samples placed in one buffer, the higher the resolution of the FFT will be. If the buffer is too large, however, a substantial time frame will be compressed; for example, if the sample rate is equals to 1kHz and the buffer length is two hundred samples, the FFT represents only five frames per one second. As such, the time representation of the spectrum suffers; hence, the balance between the size of the frame and desired resolution must be maintained. Moreover, if one decides to use a large buffer while maintaining a broad time representation, a so-called window overlap is used. The idea is to provide approximately 25% overlap between consecutive buffers (Levis, 2011).

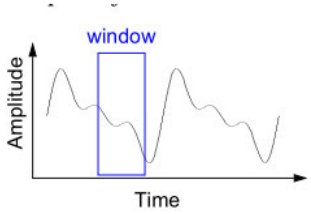


Figure 3-12 - FFT window (Source: Misiti et al. 1996)

ix. FFT for Vibration Analysis

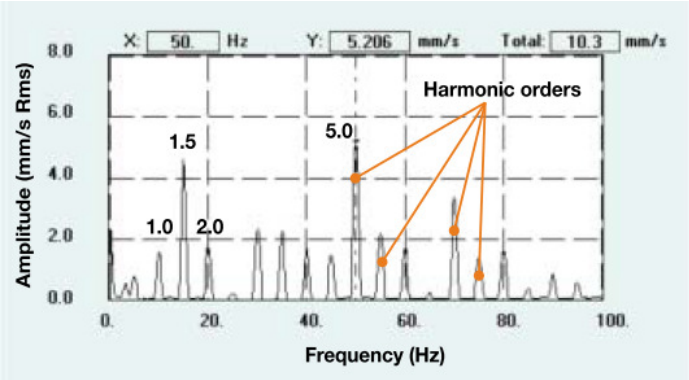


Figure 3-13 - Example of a signal spectrogram plot (Source: Tienhaara 2004)

Figure 3-13 represents a typical FFT obtained by processing the vibration signal of a running four-stroke diesel engine. The peaks on the graph represent orders of the main harmonics. The first order corresponds to the engine RPM. Additionally, it can be seen that some half orders are present. The full orders account for the mass forces, such as bending and torsion, while the half orders occur due to an imbalance in combustion gas forces. It is worth noting that the presence of particular harmonics and their magnitudes greatly depends on the number of cylinders and the arrangement of crankpins; for example, inline six or V8.

One method used to counter a mass forces imbalance is placing counterweights on the crankshaft (Tienhaara, 2004).

x. Frequency domain

The signal frequency domain is represented by plotting the real part of the output of the FFT function, as the FFT also produces complex numbers. The resultant graph provides

a baseline for the remaining calculated parameters, such as the RMS and the Kurtosis. The spectrogram visualises the base frequencies of sinusoidal components of the signal. For instance, the spectrogram of a four-stroke internal combustion engine will certainly show several distinct harmonic orders, such as the rotational frequency (first order), the firing order harmonics (third order), and some cylinder pressure variation harmonics (usually half orders of the main harmonics). By examining such a spectrogram, one can determine if any unusual harmonic is amplified on the plot. Comparing the spectrogram with values of the RMS, the decibel scale, and the Crest Factor, the user can make an informed decision about the condition of the equipment.

The torque pulses caused by combustion contribute the most to the few first harmonic orders. The combustion frequency can be estimated as follows (Li, et al., 2016):

$$f = \frac{2}{60c} ni$$

Where:

n = RPM

c = number of strokes (e.g. two- or four-stroke engine)

i = number of cylinders.

It is possible to detect fault conditions based solely on the signal spectrogram. The following figures illustrate this clearly.

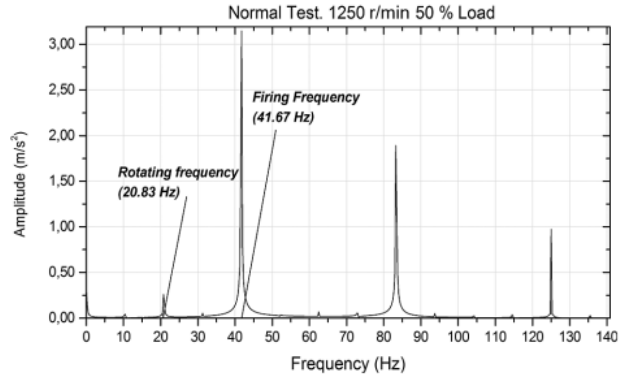


Figure 3-14 - Frequencies present in a four cylinder engine under normal condition (Source: Broatch et al. 2008).

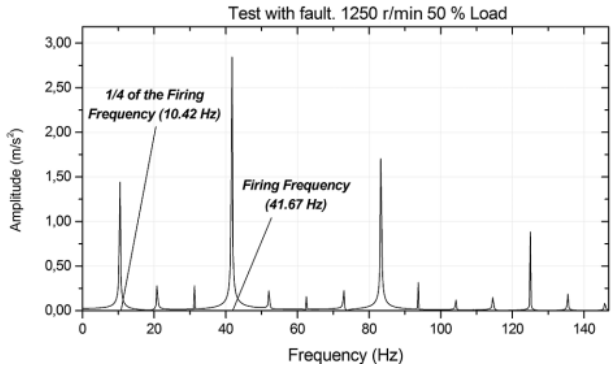


Figure 3-15 - Frequencies present in a four cylinder engine under a faulty condition. (Source: Broatch et al. 2008).

Figures 3-14 and 3-15 are spectrograms of the vibrations of a four-cylinder engine in normal working conditions and with an induced fault on a single cylinder, respectively. It is a clear sign that when the fault is present, the first order harmonic appears as a fraction of the firing frequency. Moreover, the amplitude of the first frequency increases significantly.

3.3.4 Signal Filtering

It is inevitable that a signal will be contaminated by noise. Typically, noise is introduced by the signal acquisition device itself. Frequencies from the Alternating Current (AC) supplies are often the main source of electrical noise in the system. As explained in section 3.3.3, the FFT decomposes a signal into its sinusoidal components where one or more of them may belong to noise. Knowing the frequency of the noise component makes it possible to eliminate it from the spectrum of the signal; however, since the frequency of the noise is not always apparent, several methods have been developed to aid the filtering process (Levis, 2011).

i. Discrete filtering

The discrete filter can be characterised with the pass band, the gain, and the transition bandwidth. In other words, the filter is designed to pass the desired frequencies and attenuate those that are not of interest (see Figure 3-16). The magnitude of the gain to bandwidth ratio (the slope) represents how sharp the transition between the pass state and the cut-off state is. Ideally, one would like to see a vertical, brick wall-like transition, which is not always achievable.

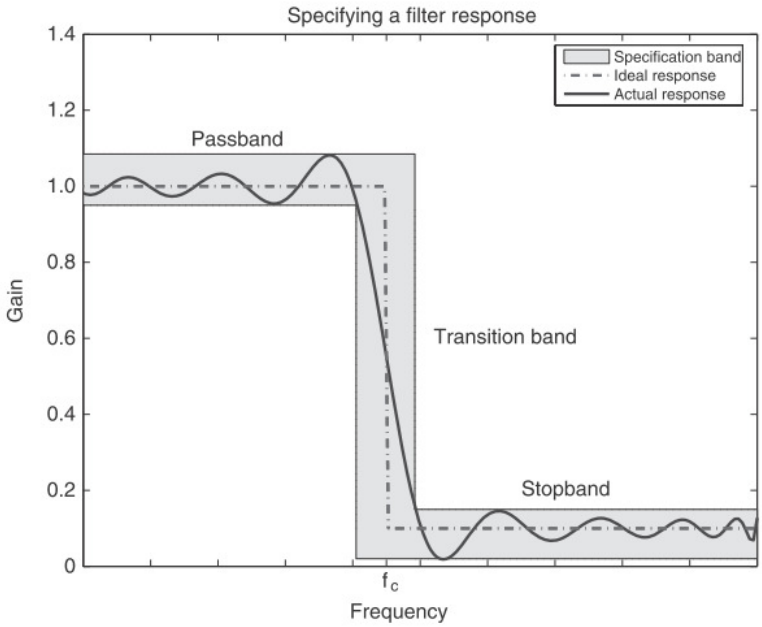


Figure 3-16 - Typical low-pass filter transition band (Source: Levis 2011).

Two main types of discrete filters can be distinguished. Low-pass filters attenuate frequencies that are lower than the cut-off threshold, whereas high-pass filters allow only frequencies above the threshold to pass. It is possible to combine both types of filters to allow only desired bandwidths to pass, forming a band-pass filter (see Figure 3-17). By bringing the thresholds of both filters closer together, one can design a filter that cuts off only a specific frequency (Levis, 2011). This type of filter is called a notch filter.

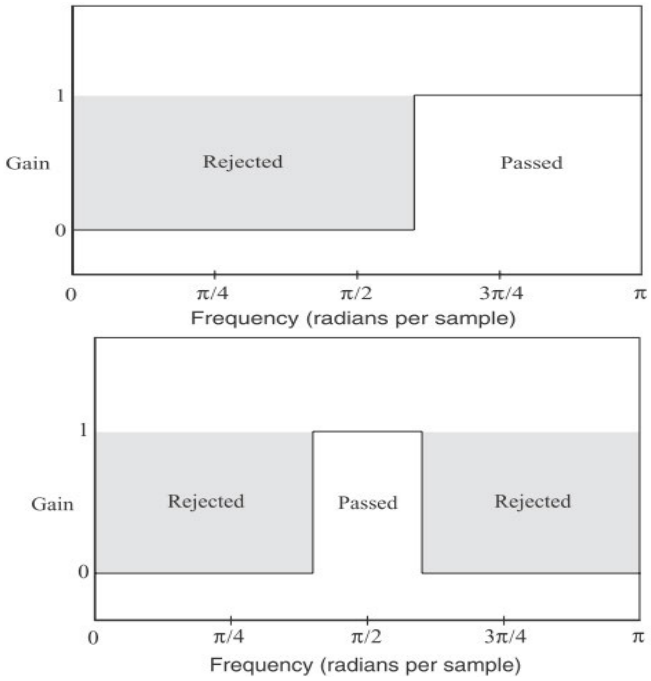


Figure 3-17 - High-pass (top) and band-pass (bottom) filters examples (Source: Levis 2011).

The main disadvantage of discrete filtering is that, if a signal is passed through a low-pass filter, and the same data is then passed through a high-pass filter to obtain the bandwidth filtering, we will end up with twice as much data as we had before the filtering (see Figure 3-18). For this reason, in the case of real-time signal processing, this method requires powerful computers to process the doubled amount of data.

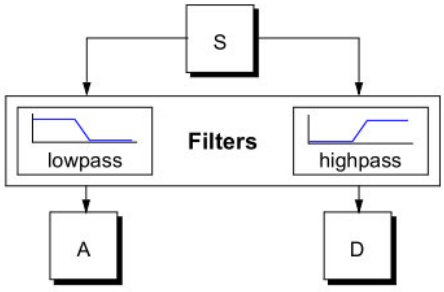


Figure 3-18 - Passing a signal through two filters produces two signals. (Source: Misiti et al. 1996).

To set the filtering thresholds, the frequency of the noise must be known. In a real situation, the source of the noise and its frequency are not always apparent; therefore, other methods of deriving the noise components from a signal were examined.

ii. Wavelet signal decomposition and filtering

Although wavelet decomposition is a relatively new technique, it has gained widespread popularity in signal processing due to its powerful potential to reveal the time and frequency information of the signal (Moosavian, et al., 2016). It has the ability to distinguish different trends in a signal, which other methods, such as the FFT, can overlook.

As noted earlier in the discussion (section 3.3.3), the FFT method utilises a kind of windowing of the signal. The window's size is chosen to balance the time density and desired resolution (accuracy) of the spectrum plot. The main drawback of this method is that the user is unable to vary the window's size during the sampling.

Wavelet analysis (WA) is the next step in signal processing. It allows the application of a flexible windowing technique, or as it is called by WA authors, scaling. In short, the wavelet is a pattern that represents various windows' sizes. For instance, one can opt to use large windows to reveal precise low-frequency components, while applying small windows for higher frequencies (Misiti, et al., 1996).

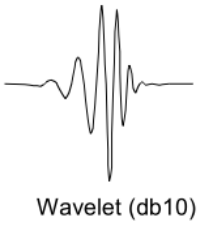


Figure 3-19 - Sample of a wavelet (Source: Misiti et al. 1996)

Figure 3-19 shows a simple wavelet pattern. The algorithm that performs the WA uses scaling and shifting of the wavelet pattern. Scaling is the stretching or compressing of the wavelet, while shifting means moving along the signal; thus, the WA algorithm selects a discrete length of the signal and compares the wavelet with the signal to obtain the correlation coefficient (see Figure 3-20).

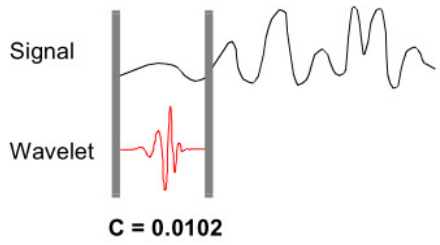


Figure 3-20 - Correlation coefficient (Source: Misiti et al. 1996).

It then shifts the wavelet to the next discrete piece of the signal (see Figure 3-21).

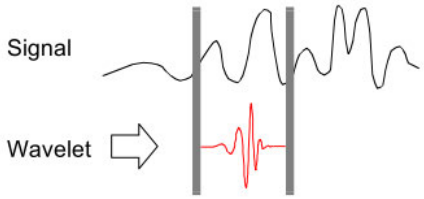


Figure 3-21 - Shifting of a wavelet (Source: Misiti et al. 1996).

After shifting to the end of the signal, the WA algorithm applies the scaling to the wavelet and repeats shifting from the beginning of the signal (see figures 3-22 and 3-23).

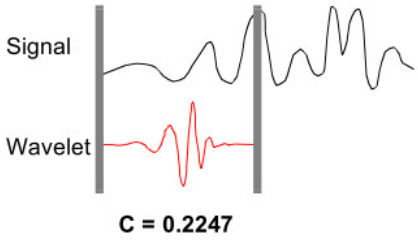
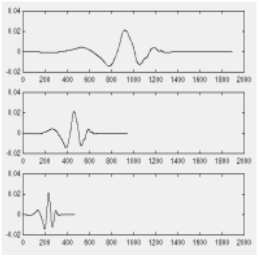


Figure 3-22 - Stretching a wavelet (Source: Misiti et al. 1996).



$$f(t) = \psi(t) \quad ; \quad a = 1$$

$$f(t) = \psi(2t) \quad ; \quad a = \frac{1}{2}$$

$$f(t) = \psi(4t) \quad ; \quad a = \frac{1}{4}$$

Figure 3-23 - Scaling of a wavelet (Source: Misiti et al. 1996).

Implementing wavelet decomposition and reconstruction of a signal enables the extraction of required frequency components from a signal, avoiding side effects such as aliasing and power leakages, which are commonly introduced by conventional filtering techniques (Geng, et al., 2003).

Various wavelets are available through the Matlab Wavelet Toolbox. Each of the wavelets provides different outcomes to the decomposition of a signal. By trial and error, it is possible to select the wavelet that will reveal desired properties of the signal. Moosavian, et al. (2016) studied the use of wavelets in detecting abnormal content in the vibration signal of a four-stroke reciprocating engine. They suggested using the "dmey" wavelet following eight levels of decomposition.

iii. Wavelet vs Discrete filtering

As discussed earlier, in section 3.3.3, the application of discrete filtering doubles the amount of data to be processed. Wavelet filtering introduces the concept of downsampling (see figure 3-24), which eliminates every second sample.

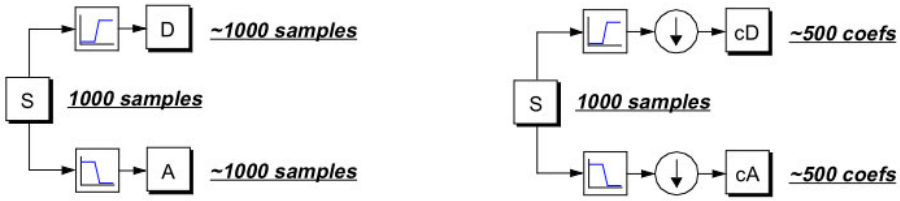


Figure 3-24 - Discrete filtering vs. Wavelet's down sampling (Source: Misiti et al. 1996).

It could be suggested that downsampling introduces an aliasing effect; however, it will be explained why this should be ignored later on.

After the initial wavelet filtering, high- and low-frequency components are decomposed into several levels (see figures 3-25 and 3-26). The user specifies the number of the levels based on the signal background information, or an experimental approach (Misiti, et al., 1996).

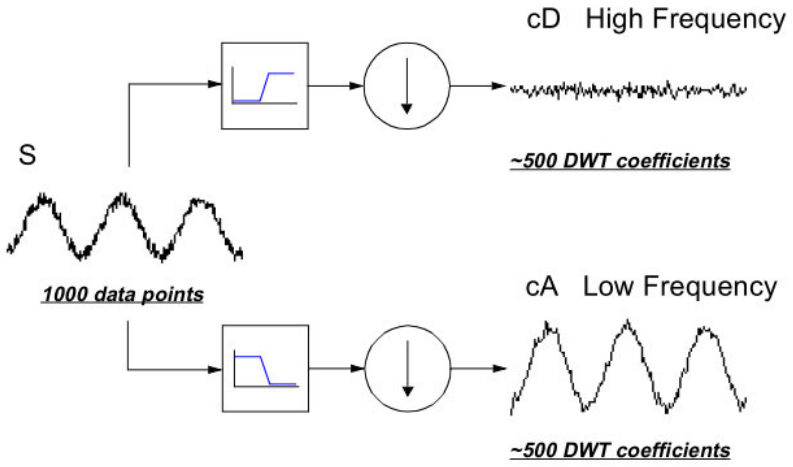


Figure 3-25 - Wavelet signal decomposition (Source: Misiti et al. 1996).

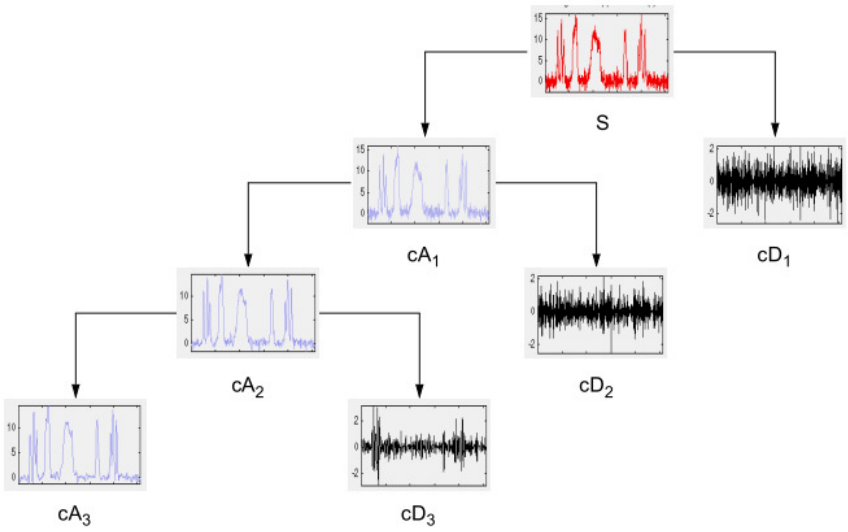


Figure 3-26 - Wavelet signal decomposition (Source: Misiti et al. 1996).

The user specifies which level (cD) is considered noise and initiates the signal reconstruction. The reconstruction process is simply a reverse of the decomposition using upsampling. This upsampling eliminates the aliasing effect (see figure 3-27).

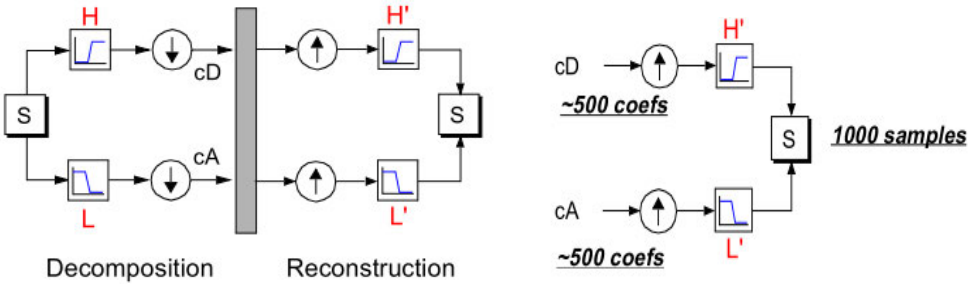


Figure 3-27 - Up-sampling and the signal reconstruction (Source: Misiti et al. 1996).

Since the WA filtering output contains the same number of samples, the computational time required for the signal's post-processing is less than if discrete filtering had been implemented.

The most popular wavelets were invented by Ingrid Daubechies, who is renowned in the wavelet research discipline.

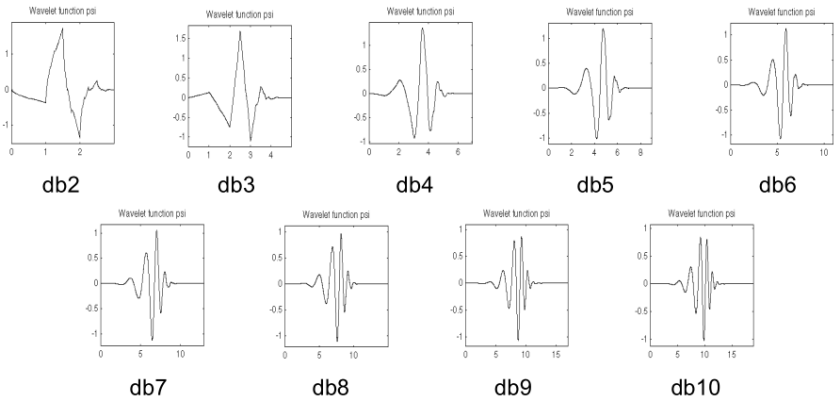
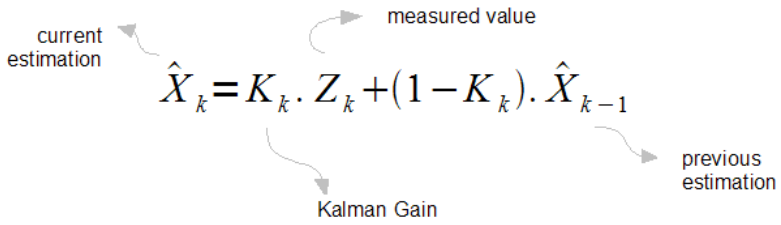


Figure 3-28 - The Daubechies's wavelets (Misiti et al. 1996).

In general, the higher the wavelet number, the more accurate the decomposition becomes; however, accuracy is achieved at the cost of computational time.

iv. Kalman Filter

The Kalman filter was introduced in 1960 by the American electrical engineer Rudolf Emil Kalman (born in Hungary). It is worth noting that in mathematical terms, the Kalman filter is not exactly a filter; rather, it is an approximation of the state of the signal between two measurements, based on statistics such as the Gaussian distribution. This method is widely accepted in smoothing noisy and intermittent signals, and gained its popularity after being used by NASA in the Apollo program. The algorithm of the filter is simple and uses little computational time.



The derivation process of the equation is rather involved and, for transient noise conditions and 2D and 3D measurements, requires an extensive knowledge of mathematics to solve a set of differential equations. For instance, if the noise changes over time, the Kalman gain coefficient must be calculated for every iteration. In the case of filtering an electric line noise, however, the problem involves 1D measurements of the line voltage. In this way, the Kalman algorithm is greatly simplified into a set of linear equations.

Time update set of equations

$$X_k = X_{k-1}$$

$$P_k = P_{k-1}$$

Measurement update set of equations

$$Kk = \frac{Pk}{Pk + R}$$

$$Xk = Xk + Kk(Zk - Xk)$$

$$Pk = (1 - Kk)Pk$$

Where:

Xk = Filtered signal.

Z_k=Unfiltered signal.

P_k= Filter sensitivity coefficient (estimated experimentally).

R = Standard deviation of the signal.

K_k=Gain factor.

The coefficient P determines the filtering effect of the Kalman algorithm on the signal. For instance, the greater the value of P, the more Kalman assumes that little noise is present. If we reduce the value of P, Kalman assumes that the greater portion of the signal consists of noise. If P equals zero, the whole signal is treated as noise.

The algorithm is explained thus. The initial value of the signal and the value of the noise intensity, P, are selected and substituted into the first set of equations. The results are passed to the second set of equations to obtain new values of Xk and Pk. These new values are then passed back into the first set of equations, and the process is repeated (Welch and Bishop, 2001).

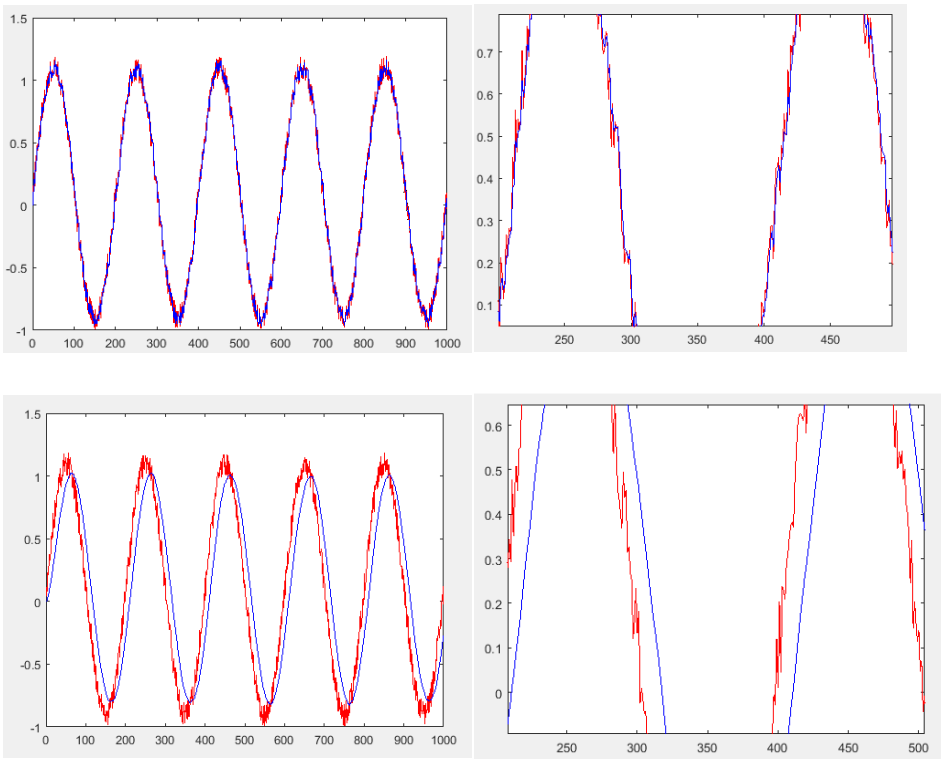


Figure 3-29 - Kalman filtering. Top plot P=1, Bottom plot P=0.1 (Produces with aid of MATLAB 2016A student version)

Matlab allows the Kalman filtering algorithm to be easily implemented for one-dimensional signals (see Appendix B for the example code). Figure 3-29 represents a sinusoidal signal contaminated by random noise (red) overlaid by the filtered signal (blue). As shown, when the value of P is 1, almost no filtering is applied, whereas when P equals 0.1, the filtered signal is almost de-noised.

3.3.5 Signal Processing Software

Matlab was chosen as the signal processing software for its flexibility to be used on any platform that has Matlab installed (Howard, 1995). Other advantages of Matlab are outlined as follows:

- The signal spectrum can be computed with very high resolution and along the full frequency band.
- The signal data can be stored in numerous file standards.
- Powerful graphical capabilities.
- The flexible graphic user interface can be easily altered to support required specifications.

3.3.6 Vibration Acquisition Hardware

The most common and widely accepted types of sensor used in vibration acquisition systems are accelerometers. There are a variety of accelerometers available on the market, from fairly expensive units with high measurement precision to relatively cheap but reliable products. There are two types of accelerometers. These are micro-electromechanical systems (MEMS) sensors and piezoelectric sensors.

1. MEMS sensors

The working principle of MEMS accelerometers is based on a mass that is suspended between two electrodes. The electrodes represent a variable capacitor (O'Reilly, et al., 2009). As acceleration occurs, the mass shifts due to inertia, causing it to move closer

to one of the electrodes (see Figure 3-30). As a result, the capacitance changes, leading to variations in the output voltage. The main advantage of this type of sensor is that they have in-built amplifiers and low-pass filters; therefore, the output voltage can be sent directly to a microcontroller. Some MEMS sensors have an analog-to-digital converter and can communicate via Inter-Integrated Circuit (I2C) or Serial Peripheral Interface (SPI) interfaces, which makes them adaptable to any embedded systems. One of the main disadvantages of MEMS sensors is their narrow bandwidth. This means that they are not suitable for detecting vibrations consisting of a wide range of frequencies.

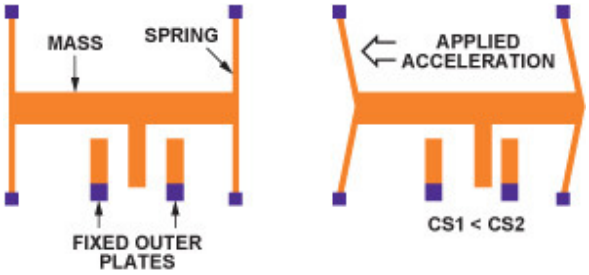


Figure 3-30 - Schematic representation of a MEMS accelerometer (O'Reilly et al. 2009)

2. Piezoelectric Sensors

These kinds of sensors are similar to MEMS. The only difference is that the mass that moves due to inertia acts on a piezoelectric crystal, causing it to generate a potential difference in millivolts (see Figure 3-31). The main advantages of piezoelectric accelerometers are as follows:

- The MEMS provide the output of a real-time vibration in 'g' units, whereas piezoelectric accelerometers are able to measure the frequency of change of the acceleration in Hz, which makes them more suitable for vibration reading.
- They are extremely responsive and possess very wideband sensitivity, ranging from 1Hz to tens of kilohertz.

- They do not require an external power supply.

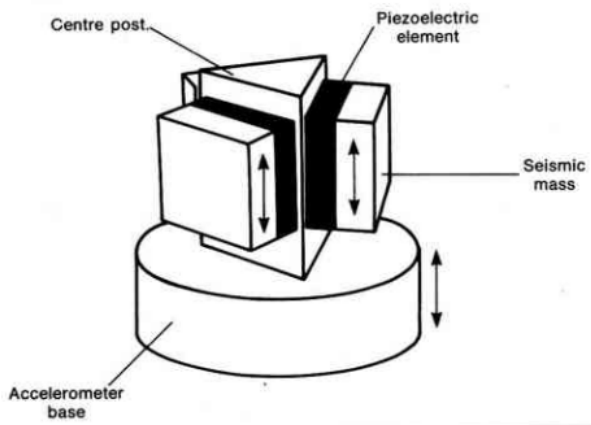


Figure 3-31 - Schematic of a piezoelectric accelerometer (Serridge & Licht 1987)

.

Table 3-1- Accelerometers considered for the project (Source : www.analog.com and www.endevco.com)

Accelerometer	type	Voltage	Range	Output	Bandwidth
ADXL335	MEMS	1.8-3.6	3.6 g	Analog	0.5-1600Hz
ADXL345	MEMS	2.0-3.6	2-16g	SPI/I2C	0.05-1600Hz
ENDEVCO	Piezoelectric	N/A	1-1000g	Analog	1-10000Hz

Table 3-1 provides the technical characteristics of the most common accelerometers available of the market.

3. Microcontroller

One of the major objectives of the project is to design an embedded system that is capable of making simple, logical decisions based on inputs from sensors and processed data from the host Personal Computer (PC). Sensors of pressure and

temperature, for example, provide analog outputs; therefore, availability of an analog-to-digital converter (ADC) module is paramount.

The selected microcontroller should have the following specifications:

- 20 as the minimum number of Input/Output (I/0) pins.
- An in-built ADC.
- A programming language converter.
- Wide support from a community of enthusiasts.

There are a few distinct platforms available, such as the AVR®, the Microchip®, and the AMR®.

Raspberry Pi 3

Raspberry Pi 3 (see figure 3-32) is a standalone computer, not a microcontroller. It has a powerful central processing unit (CPU) and a graphics processing unit (GPU), which are both governed by a Linux-based operating system. These features make Raspberry Pi 3 an ideal platform for an independent system that can acquire signals, perform real-time processing of the signal, and make logical decisions following outputs. The project's signal processing, however, is based on the Matlab environment, which is currently not supported by Linux-based systems. Nevertheless, similar mathematical packages are available for Linux platforms; thus, the Raspberry Pi 3 could be considered as the autonomous module to perform the signal acquisition and processing in future work.

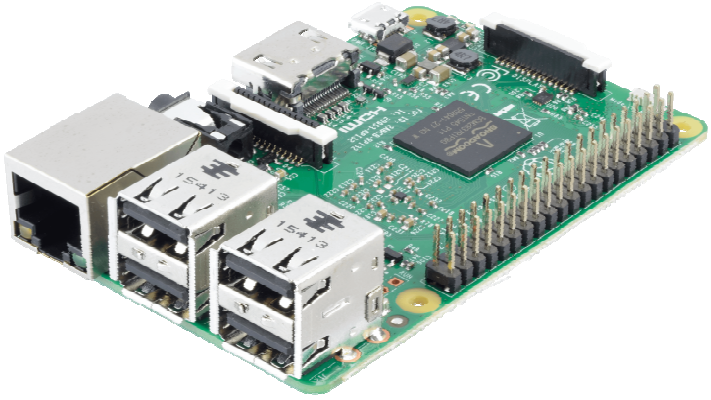


Figure 3-32 - Raspberry Pi 3 board (Source: www.raspberry.piaustralia.com.au)

Microchip PIC®

The abbreviation PIC® stands for Peripheral Interface Controller. The first PIC microcontroller was issued by General Instruments (later Microchip Technology) in 1975. Since then, the company has developed many different packages suitable for various applications. PIC microcontrollers are available with various clock speeds, in-built communication interfaces, and I/O pins. They gained widespread popularity in industrial applications for their broad functionality and wide range of specifications. The PIC can be programmed with the aid of the MPLAB© IDE, which is available on the Microchip Technology Inc website. The MPLAB has a dedicated C compiler and supports entries written in assembly language. It also has a de-bugger, which allows the states of variables to be observed and the breaking points in the main code to be set.

The main drawback of a PIC platform is that the registers addressing commands and the ports settings are not unified across different PIC models; hence, there is a lack of written libraries to communicate with peripheral devices. For this reason, the user is required to directly program registers and ports of microcontrollers, which are not simple tasks and necessitate consultation with the PIC datasheets. Despite these outlined issues, PIC microcontrollers are widely used in industrial application. This

being said, they are not quite suitable for the research purposes of this project, where working solutions should be easily prototyped and tested.

Arduino

The Arduino originated in 2003, in the Interaction Design Institute Ivrea of Italy, as an easy-to-learn platform for students. It provided a standard interface to connect a microcontroller to various sensors and actuators. In the following few years, the project became widespread amongst electronic designers and hobbyists. Nowadays, it has a large community, which constantly participates in the evolution of the platform. The Arduino platform has a standardised circuit board and I/O pin layout, making it possible for many enthusiasts and manufacturers to produce extension boards called shields (see figure 3-33). The shields can be attached on top of the Arduino board to add more functionality to it. An enormous variety of different shields, compatible modules, and sensors for almost any task are available for sale. The generic Arduino boasts common communication protocols such as Universal Asynchronous Receiver Transmitter (UART), I2C, and SPI.

The software development environment is supported by dedicated IDE, which uses C programming language. One of the disadvantages of Arduino is the absence of a de-bugger, which makes tracing errors in code a less than trivial task. Some developers, however, such as Autodesk, provide an alternative environment with simulation capabilities. The main advantage of the Arduino IDE is that it is supported by many preassembled libraries to communicate with various peripherals, which makes programming intuitively understandable and straightforward. Based on the information above, it can be concluded that Arduino-based boards provide an excellent platform for the proposed system prototyping.

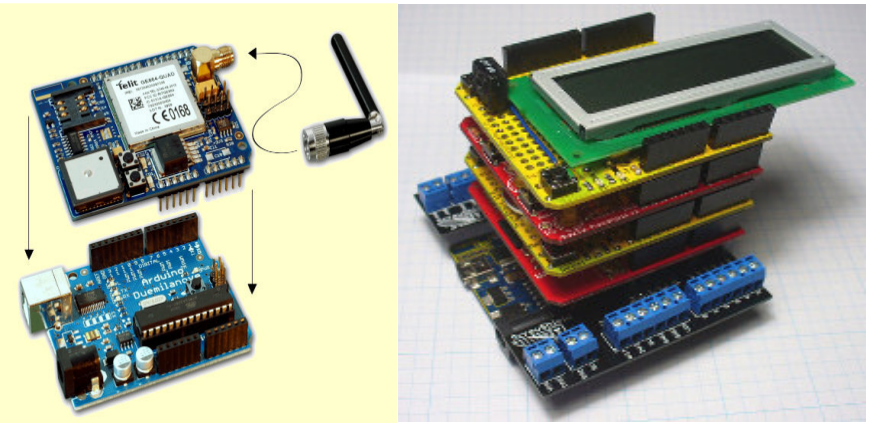


Figure 3-33 - Arduino board and stackable shields (www.gravitech.us).

Table 3-2 - Microcontrollers specification (Source: www.atmel.com, www.microchip.com, www.raspberi.com)

Platform	Arduino	PIC18	Raspberry Pi 3
Core	AVR	Microchip	ARM
Clock	16MHz	8-20MHz	1.2Ghz
Word's length	8/16 bitst	8/16/32 bits	32/64bits
Flash memory (kb)	256	32	1Gb
I/O pins num	52	40	40
I/0 logic voltage	3-5V	5V	5V
Operating system	N/A	N/A	Linux based
Programming Language	C	C	C, Java, Python
ADC	10 bits	10 bits	N/A
Cost	$10-30	$10-40	$30-60

Table 3-2 lists the key features of the discussed platforms.

3.4 Conclusions

Recent studies have proven that monitoring wear debris in the lubrication oil in combination with vibration analysis provides comprehensive information on the current state of a machine (Peng, et al., 2005). The utilised advantages of both methods complement and strengthen the system's ability to detect failure (Yibo, et al., 2010; Li, et al., 2012). The test prototype, therefore, was chosen to be able to detect wear particles in the oil by applying the pressure drop method, and use vibration analysis as the secondary source for the decision-making algorithm.

It was decided that, for the vibration analysis, the Matlab environment provides the most appropriate solutions. The signal FFT, spectrum analysis, RMS, and Crest Factor were chosen for examination as parameters that potentially reveal faulty conditions. De-noising algorithms based on discrete filtering, Kalman filtering, and wavelet decomposition were implemented.

The family of Arduino microcontrollers were selected for construction of the signal acquisition system due to their excellent compatibility with the majority of external hardware, as well as the vast support from a community of enthusiasts. The ADXL335 MEMS accelerometer was chosen as the vibration-sensing unit due to its availability, low price, sufficient characteristics, and direct compatibility with Arduino boards.

In the following chapter, the project methodology will be discussed in detail, outlining each of the design criterion mentioned above.

Chapter 4 Project Methodology

4.1 Introduction

The entire project was divided into the following stages:

1. Design the hardware for signal acquisition.
2. Write a Matlab program and GUI to process and manipulate the vibration data.
3. Acquire all necessary resources (see Table 3-1).
4. Test and adjust the signal acquisition hardware and Matlab software.
5. Fabricate the test bench, and install the test engine with all supporting subsystems.
6. Test the system.
7. Analyse the outcomes and draw conclusions.

4.2 Objectives and Scope

The scope of the project consisted of four primary objectives. The first objective was to research the ways of early detection of catastrophic failures in industrial engines. The second goal was to develop and build a working prototype of a real-time oil condition and vibration monitoring system to detect large wear particles generated by abnormal wear of crankshaft journal bearings, and detect changes in the vibration pattern. The system comprises two main sections. The first one detects the presence of large wear particles in the oil by sensing the pressure difference across a filtering element. The

second section reads a vibration signal from an accelerometer and sends it to Matlab for processing.

The third objective was to build a test rig based on a passenger car engine (preferably a Ford Falcon 6 inline engine). The test rig had to support a fixture to the test engine and the subsystems, such as cooling and fuel systems, as well as contain all the necessary sensors and gauges, such as oil pressure, coolant temperature, and RPM.

The final aim of the project was to evaluate whether the system would meet the expected performance criteria, to be achieved by conducting a series of experiments with the installed prototype. The condition monitoring system would sense and process vibration data, in addition to the oil pressure difference across the control volume. All parameters were recorded for further examination and post-processing.

The aim would then be to find correlations between the change in oil differential pressure, which is assumed to indicate the start of a fault, and changes in vibration signatures, such as a shift in harmonic orders, change of the RMS, and change in the Crest Factor of the signal.

4.3 The GUI interface

Matlab was chosen as one of the methods of calculation and visualisation of the data. To provide an interface to call different subroutines and display their output, a GUI program was written (see figure 4-1). The main functionality of the GUI is to allow communication between Matlab and the signal acquisition device, enable/disable signal filters, plots and post-processing functions, and provide data-recording and replaying features. The student version of Matlab, the Digital Signal Processing (DSP) Toolbox, and the Wavelet Toolbox were used to write the GUI script. The DSP Toolbox was included in the student license package; however, the Wavelet Toolbox had to be purchased and installed separately.

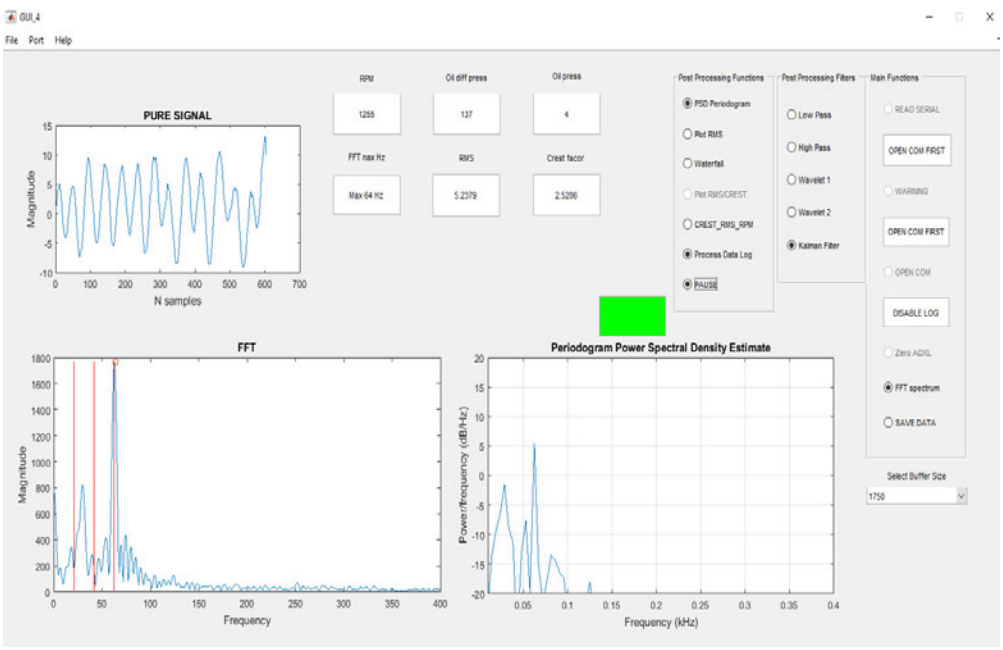


Figure 4-1 - The main window of the GUI (Produced with the aid of MATLB 2016A student version).

4.3.1 The GUI Program Structure

The main script of the GUI interface allows data to be processed in both the online and the offline modes. Figure 4-2 provides the GUI script logic chart that describes the flow of data within the main program.

The online mode begins by establishing communication with the signal acquisition hardware. The user then initiates the data reading process. The received data goes through the data-sorting algorithm, where the vibration data becomes separated from the sensor readings. After separation, the data is available for the output functions.

The offline mode starts with the user selecting the recorded data and enabling offline data processing. The data-sorting algorithm here is different to the online mode of data sorting; however, the signal filters and output functions are the same for both modes.

By activating corresponding radio-buttons, the user determines whether the data undertakes filtering, or is immediately sent to the output functions. The user can also chose which output function to enable.

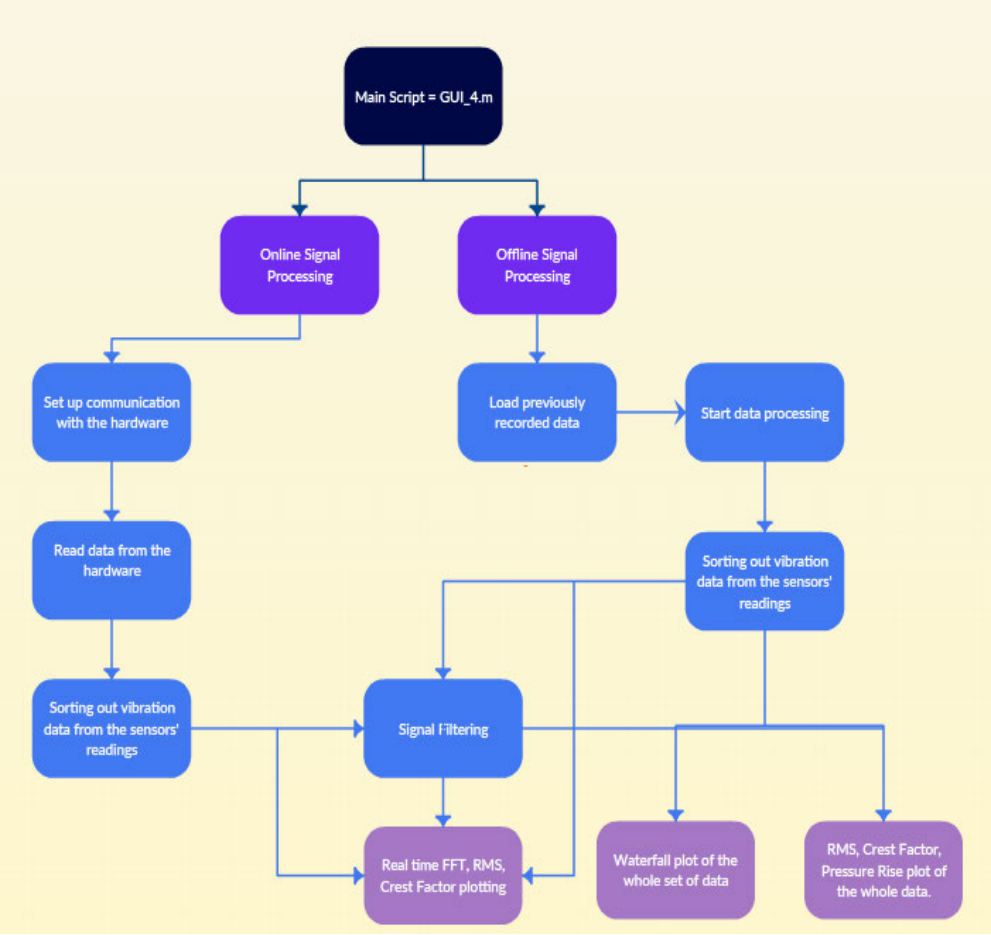


Figure 4-2 - The GUI script logic chart.

4.3.2 The GUI Toolbar

The top toolbar of the GUI contains three menus (see Figure 4-1). The first "File" initiates the window where log files can be selected for replay and post-processing. By clicking on the "Port" menu, the program can be told in what Communication port (COM) the data-logger is connected. The "Help" menu opens the simplified user manual in *.txt format.

4.3.3 GUI Functions

The GUI made acquiring, processing and analysing of the data a straightforward and intuitive process. By employing radio buttons, the user can insert desired functions and filters into the signal processing chain without needing to manually alter the Matlab script.

The control buttons are grouped into three categories: the main function, the post-processing filters, and the post-processing functions. The following sections will outline the functionality of each panel.

4.3.4 Main functions.

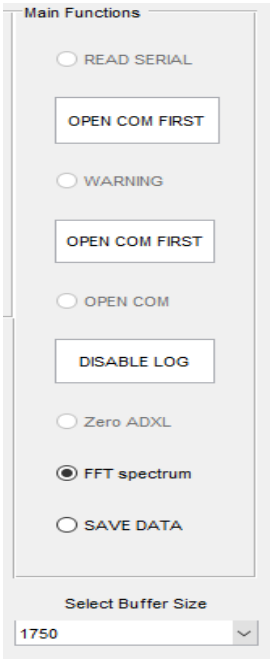


Figure 4-3 - The "Main Functions" group.

The "Main Functions" group (see Figure 4-3) collects all functions that are necessary to start the online signal acquisition and processing. As shown, some of the radio buttons appear in grey to signify that some of the functions cannot be executed if the preceding functions are not active. The dialog boxes located below the buttons display guidance messages. For instance, the "OPEN COM" button will not be active when offline data processing is active. In this case, the user must stop the data log processing before the system allows connection to the signal acquisition hardware.

The pop-up menu, "Select buffer size", is used to select the size of the data buffer (see Figure 4-4). The data buffer determines how many characters in the string of data Matlab receives.

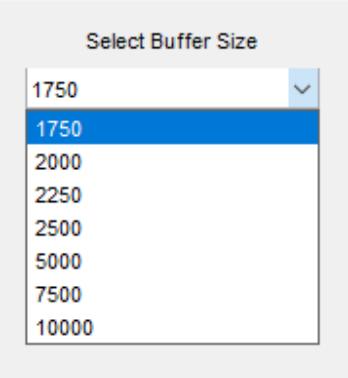


Figure 4-4 - The buffer size selection popup window.

The larger the buffer, the denser and more precise the FFT that can be retrieved from the signal is. The drawback of the large buffer is that it affects the latency due to the limited sampling rate of the signal processing microcontroller; hence, the larger the gap between the readings becomes. Specifications of different microcontrollers vary. As such, for powerful models, the buffer size can be increased. After several experiments, the buffer size of 2500 characters was decided upon. If the user does not specify the buffer size, the system will apply a buffer of 2000 and display a warning message.

"OPEN COM" initialises the communication between Matlab and the signal acquisition hardware. If this button is active, the Matlab sends a series of test commands to the hardware to provide the pre-start initialisation. If one tries to enable the button with no device connected to the COM port, the system will display a warning message.

"ZERO ADXL" - Since the accelerometer sensor can be situated at various locations, it is necessary to zero-reference the readings to the current position of the sensor. The data reading cannot be started unless this calibration is accomplished. During the calibration process, the "READ SERIAL" button is disabled. The calibration script sends the request to the hardware, which then launches an internal calibration subroutine (see Appendix D for the source code) and returns the value that is used by the program to zero-reference the signal. At times, it is necessary to repeat the sensor calibration in order to obtain the zero-centred waveform.

"READ SERIAL" enables the online signal reading. The program uses the "fread" function to request data from the hardware. The hardware sends two separate parcels of

readings, which the program receives as a string array. The first parcel contains the vibration data, followed by the sensor readings. The hardware sends a delimiter character between the parcels of data. Through the "regexp" function, Matlab then splits the string into two blocks: the vibration signals and the readings from the sensors. Finally, implementing the "cell2mat" and "eval" functions, the data is converted into integers for further processing.

"FFT Spectrum" performs the FFT transformation of the signal. The "fft" function is used to obtain the Fast Fourier Transformation of the signal. The GUI then plots the real part of the FFT as the signal spectrum. See chapters 3 and 5 for examples of graphs produced by this function.

"SAVE DATA" starts recording the data. By using the "dlmwrite" command, the data is saved for later analysis. See section 3.3.7 for further explanation.

"WARNING" is used as the emergency stop function of the test engine. If the "WARNING" button is active, the script sends an alert signal to the microcontroller, which calls for an emergency shutdown.

4.3.5 Post-Processing Filters

The post-processing filters can be manually inserted into the signal processing chain to reveal the features of the signal that could be obscured by unwanted components, such as noise. For detailed discussion on the filtering algorithms, see Chapter 3. The current version of the GUI includes the following filters (see Figure 4-5).

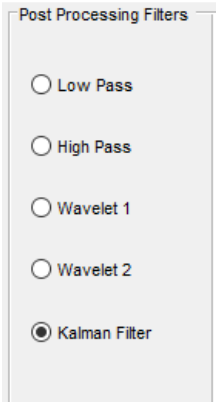


Figure 4-5 - The post processing filters group.

- Low-pass and High-pass filters

Both of these filters were designed with the aid of the DSP System Toolbox 9.2.
The "fdesign.lowpass" function was used and combined with the Butterworth (for more information regarding the Butterworth method, refer to the DSP Toolbox documentation) method to generate the "LOW_PASS.m" function. This filter cuts off frequencies that are above the specified window of 200Hz.

The "fdesign.highpass" function and the Chebyshev method (for more information regarding the Chebyshev method, refer to the DSP Toolbox documentation) were used to generate the "HIGH_PASS.m". The cutoff frequency range is below the 5Hz band.

As discussed in Chapter 3, the high-pass and low-pass filters require substantial computation power; hence, avoiding use of these filters on slow machines is recommended.

- Wavelet 1 and Wavelet 2 filters

The wavelet filters were designed with the aid of the Wavelet Toolbox. Both filters use three-level decomposition techniques and Daubechies's Wavelets. Wavelet 1 provides more aggressive de-noising than Wavelet 2, as it uses the higher level of the Daubechies's Wavelets. The first filter responds to the "DENOISE.m" function that was generated with the toolbox. The second filter is integrated into the script by means

of the "wavedec" (signal decomposition) and the "wrcoef" (signal reconstruction) functions. See Chapter 3 for detailed discussion regarding the wavelet decomposition method.

- Kalman filter

The Kalman filter has almost no effect on the computational time while producing high-quality de-noising of the signal (see Chapter 3). The algorithm was written as the "KALMAN.m" function.

4.3.6 Post-Processing Functions

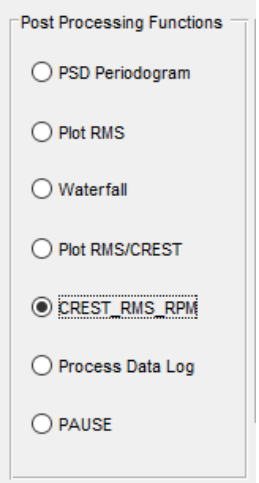


Figure 4-6 - The post processing functions group.

The post-processing functions (see Figure 4-6) allow manipulation of the real-time outputs and recorded data. The "Plot RMS/CREST" and "PSD Periodogram" functions cannot be used together since they share the same graphing space. In order to see the RMS/CREST plot, one has to disable the Power Spectral Density (PSD) function.

- PSD Periodogram

The Power Spectral Density Periodogram (PSDP) outlines the frequencies that contribute most to the signal power content (see Figure 4-7). For instance, it is evident here that the 95Hz component, which is the firing order harmonic, and the 32 Hz component, the rotational frequency, account for the majority of the noise generated by the engine.

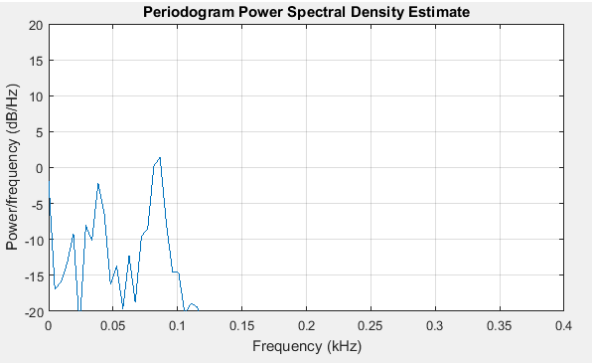


Figure 4-7 - The power spectral density plot of the signal obtained from a running engine.

The PSDP is calculated using two functions, the "spectrum.periodogram" and the "psd" functions. The first performs the FFT decomposition of the signal, passing the parameters to the second function, which calculates the PSD.

- Plot RMS

This button enables/disables the real-time RMS plot (see "Plot RMS/CREST" section).

- Waterfall

This button generates the waterfall plot of the FFT decomposition in a separate window. The "spectrogram" function is used to calculate all necessary values to be passed to the "mesh" function, which produces the waterfall plot. Initially, the plot opens in 2D mode; however, the 3D mode is also active, enabling the user to rotate the plot. Moreover, one can activate the "Wavelet 1", "Wavelet 2" and "Kalman filter"

buttons to apply the corresponding filters to the data. See Chapter 3 for a detailed description of the waterfall plot.

- Plot RMS/CREST

This button enables a real-time plot of the RMS and Crest Factor. One can exclude the RMS data from the plot by deactivating the "Plot RMS" button. The script implements the "rms" and the "peak2rms" functions to derive the corresponding values. See Chapter 3 for an example of the plot.

- CREST_RMS_RPM

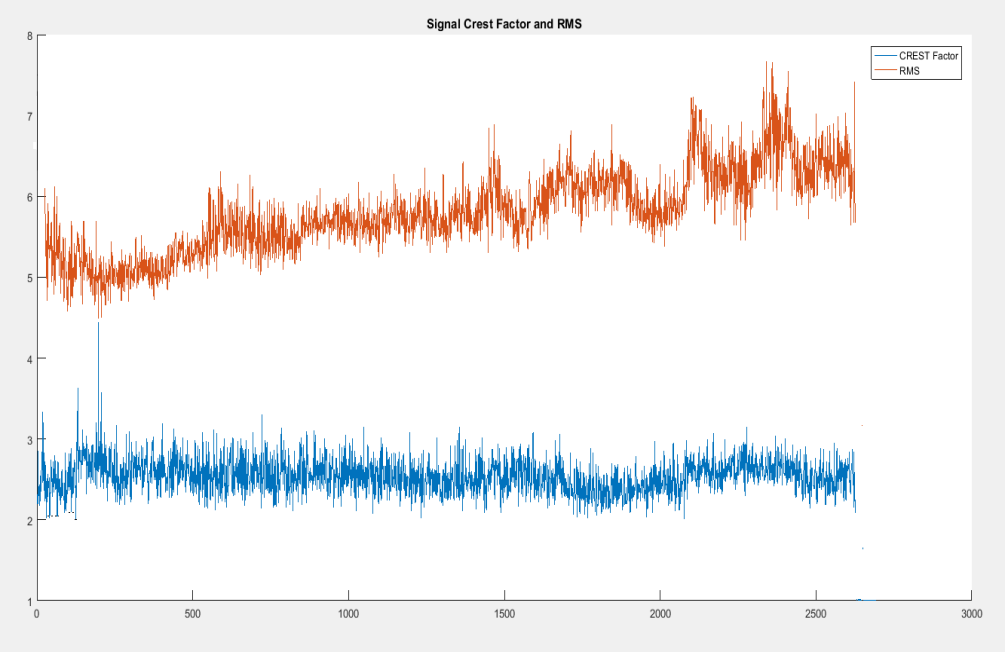


Figure 4-8 - The plot of the history changes of RMS and the crest factor (Produced with the aid of MATLB 2016A student version).

Figure 4-8 shows the parameters of RMS and the Crest Factor. The figure is automatically generated when the user activates this button.

- Data log processing

This button enables processing of the previously recorded data. Firstly, the user must select the data file from the menu by clicking "file" on the main toolbar. If no file is selected, the system will issue a warning message. All previously mentioned filters and functions can be implemented for the logged data. The script uses the "csvread" function to access the stored data.

- PAUSE

The pause button allows the logged data processing to be frozen at any time, resuming as required.

4.3.7 Outputs

The Matlab script produces the following outputs and displays them on the PC screen:

- The pure signal wave form.

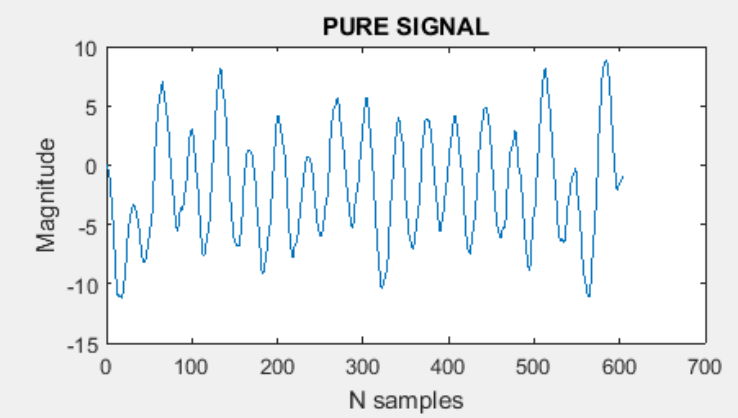


Figure 4-9 - The pure signal wave form (Produced with the aid of MATLB 2016A student version).

Figure 4-9 shows the signal waveform of 600 samples taken at the rate of 2900 samples per second.

- FFT transformation and the signal spectrum graphical representation.

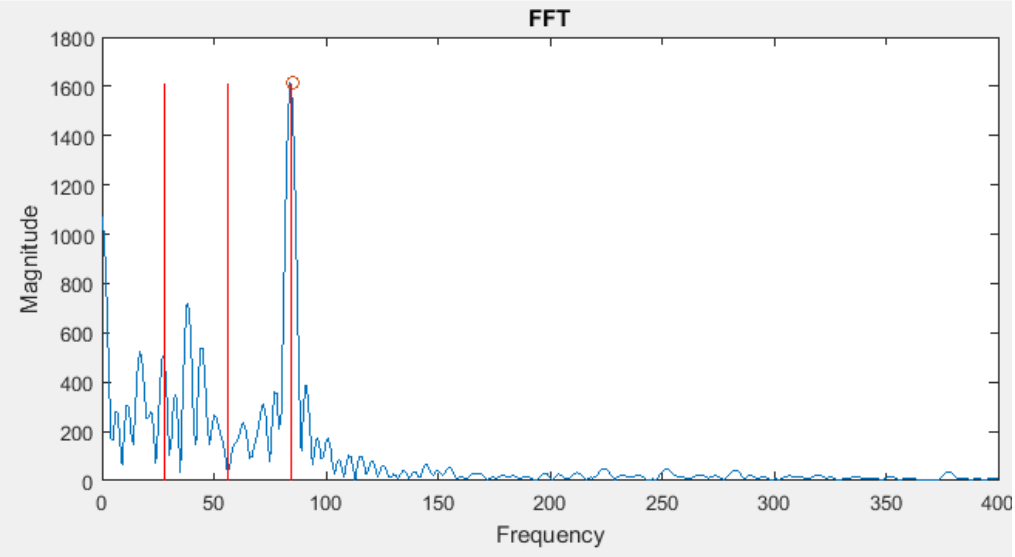


Figure 4-10 - FFT signal spectrum. The red vertical lines represent RPM/60, 2*RPM/60, 3*RPM/60 respectively (Produced with the aid of MATLB 2016A student version).

See Chapter 3 for detailed explanation of how the FFT plot is generated.

- RPM, oil differential pressure, engine oil pressure, FFT max Hz, RMS and Crest Factor values of the signal.

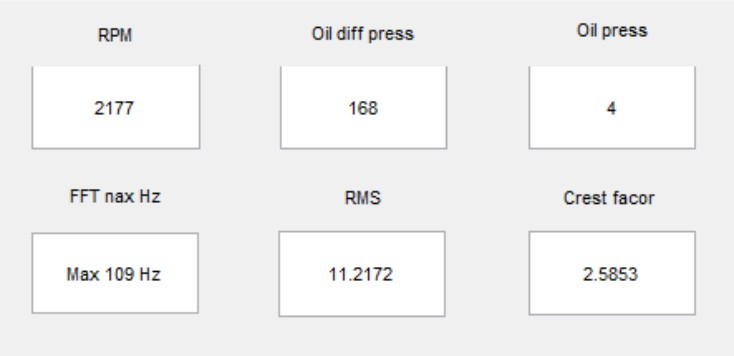


Figure 4-11 - GUI output windows.

The GUI provides several output windows for displaying numerical parameters of RPM, the debris detection pressure sensor, the engine oil pressure sensor, the maximum frequency of the FFT, RMS and the Crest Factor.

- Real-time RMS and Crest Factor plot

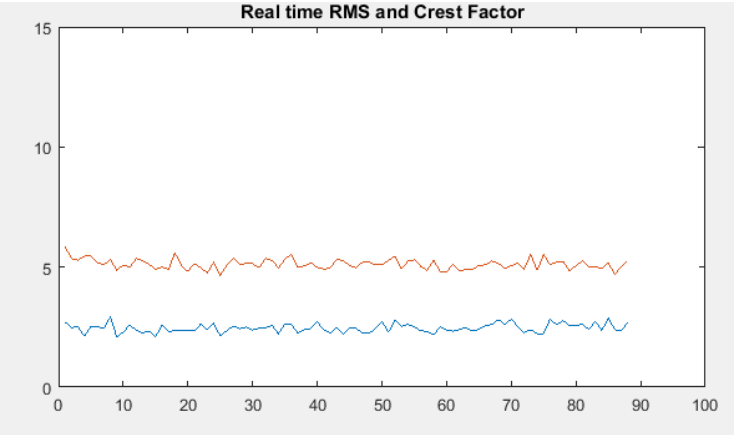


Figure 4-12 - The real time RMS (orange) and the crest factor (blue) plot (Produced with the aid of MATLB 2016A student version).

It is worth noting that, since this plot is drawn in real-time, it is impractical to include legends into the plot area. If legends are inserted, the program slows down considerably. See section 3.3.5 for the list of functions used to produce the above plot.

4.3.8 Data logging

The most helpful feature of the GUI is that it allows the data to be recorded as a *.csv file. These *.csv files can be opened and replayed in real-time, rendering this a powerful tool for comparing different readings and determining similarities or discrepancies in the signals. Numerous filters and outputs aid revealing trends in the signals. The features available for post-processing of the signal can be used to establish thresholds between signals produced by healthy and failed equipment.

4.4 Signal Acquisition Hardware

Following the design criteria, the signal acquisition hardware was designed and assembled. The hardware performs the functions of obtaining the vibration data and parameters, such as the engine RPM, engine oil pressure, and pressure rise across the debris detection system. While sorting and routing the data to the Personal Computer (PC), the hardware also accepts the commands that govern data acquisition and control the test engine.

The design criteria require that the system's hardware is able to perform the following functions:

- Vibration acquisition.
- Engine oil pressure reading.
- Reading oil pressure rise across the debris detection filter.
- Reading the test engine's RPM.
- Communicating with the host PC.
- Sending data to the host PC.
- Stopping the engine based on the programmed conditions.
- Stopping the engine based on commands from the host PC.

Based on the selection criteria (see section 3.3.6) the proposed hardware consists of two Arduino microcontrollers. The controllers communicate with each other through I2C protocol in the master/slave configuration. An ADXL335 three-axis accelerometer is used to read the vibration signal. The accelerometer is connected to the master controller, which prioritises the data processing. The master controller requests the data from the slave controller buffer only after the completion of the vibration reading cycle. This arrangement allows the master controller to be independent of the interrupt-driven data acquisition of the slave controller.

4.4.1 The Master Controller Algorithm

The master controller continuously reads the data from the ADXL sensor. After each cycle, it requests the sensor readings from the slave controller. The main task of the master controller is to collect data and transmit it to the host PC while reacting on commands sent from the PC. The master controller accepts commands that initiate communication, run the accelerometer calibration routine, and begin data transmission. In addition, if the master controller receives certain commands that are addressed to the slave controller, it passes them to the recipient.

4.4.2 The Slave Controller Algorithm

The slave controller is responsible for reading the RPM, which is provided by the Hall Effect sensor. The Hall Effect sensor is connected to one of the interrupt pins. The oil sensors and the ignition relay control wires are also attached to the slave controller, and collected data is stored in the buffer. The reading of the sensors occurs in an endless loop; however, when the RPM sensor provides an impulse to the interrupt pin, a special subroutine becomes engaged. This routine calculates the time since the last RPM

interrupt event to compute the current RPM reading. When the execution of the interrupt routine is completed, the main program returns to query the sensors.

4.4.3 The Communication Algorithm

The communication and transmission algorithms are as follows. Firstly, the master controller transmits a block of 500 comma-separated readings of the vibration data. It then sends a request to the slave controller. Following the request, data is transferred from the slave controller to the master controller. The master controller transmits the sensor readings to the PC, separating each reading by the delimiter character. The structure of the communication logic is depicted in the chart in Figure 4-13.

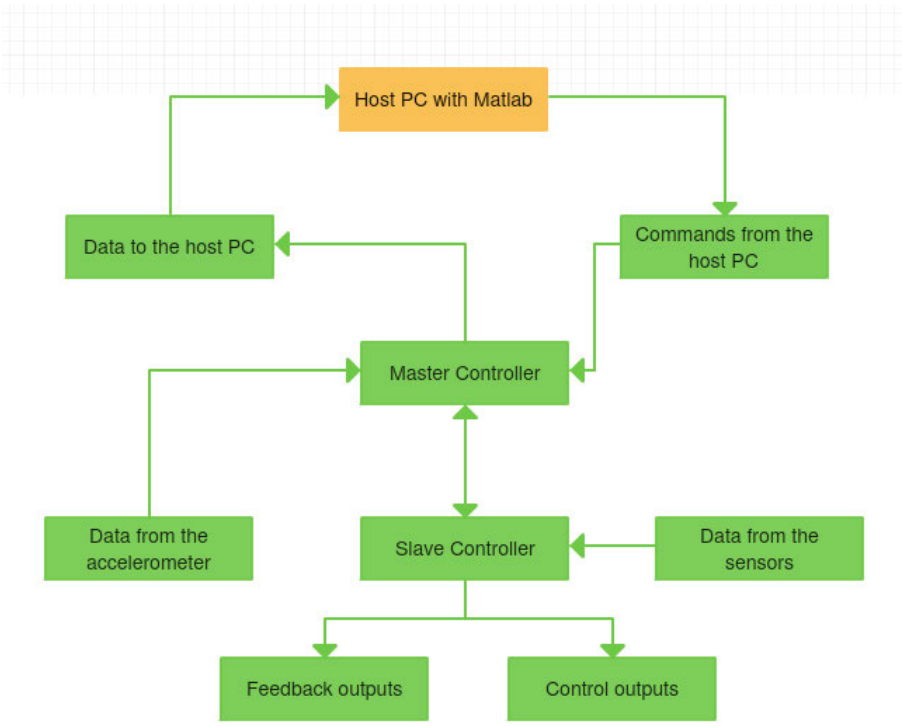


Figure 4-13 - The structure chart of communication logic between the host PC and the controllers

4.4.4 The Hardware Design

Figure 4.14 shows the connections schematic for both microcontrollers and their peripherals. The yellow and green wires enable the I2C communication. The red and black wires are connected to power supplies of two different voltages: 5 volts and 3.3 volts.

The controllers use the 5-volt supply obtained from the Universal Serial Bus (USB) cable plugged into the Arduino UNO (left). The Arduino Mega (right) uses 5 volts transmitted via the cross connection of the 5V and the GND pins. The ADXL335

requires 3.3 volts supply, so it is connected to the corresponding pin of the Arduino UNO. To connect the reference voltage to the UNO's internal ADC, the "AREF" pin of the UNO is shorted using the 3.3-volt pin, as well.

Three LEDs provide general feedback to the user from the microcontrollers. The blue LED flashes whenever RPM triggered interrupt occurs, which provides a visual signal that the system is detecting the RPM. The red LED becomes active if the pressure rises across the debris detection system exceeding the pre-programmed threshold. The green LED is lit when the stop command is received from the host PC.

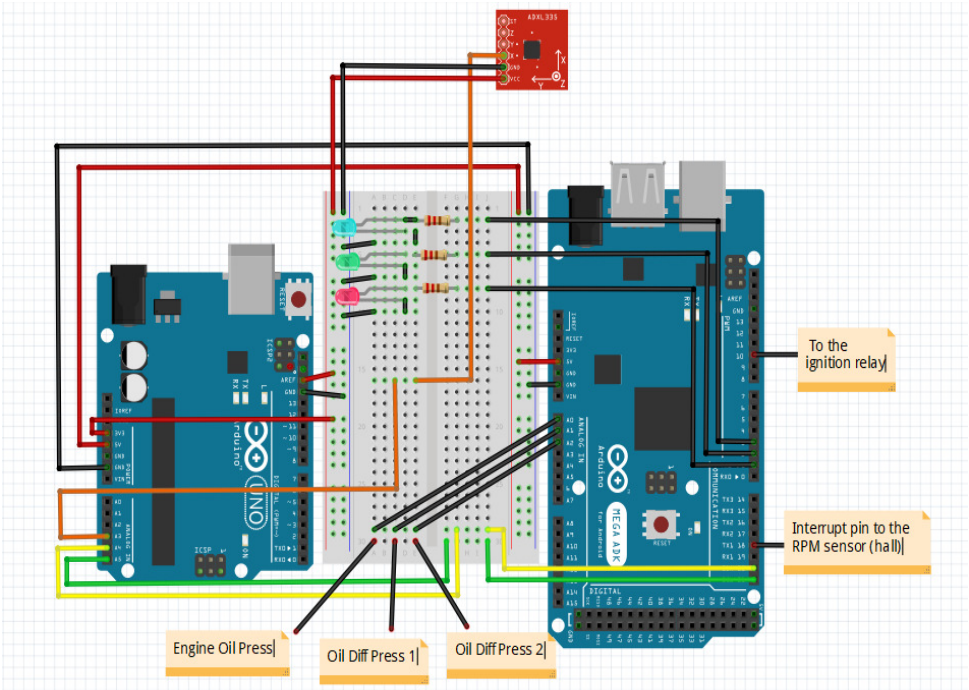


Figure 4-14- The hardware connections schematic (Produces with aid of Fritzing v0.9.3, www.fritzing.org)

4.5 Resources Requirements

Most of the required hardware was sourced from local suppliers or online. Two car engines were purchased at the local dismantlers for the test rig. The author's employer provided access to failed engine and component samples, as well as all necessary supporting information. Table 4-1 below outlines the total cost of components purchased for the project, which was self-funded.

Table 4-1 - Required resources and estimated cost.

Stage	Item	Quantity	Source	Cost
Build prototype.	Pressure senso	2	Used from work	Nill
	Accelerometer	1	Ebay	$10
	Processing MCU	1	Ebay	$50
	Oil pump with	1	Ebay	$150
	Hardware	N/A	Local stores	$200
	Engine oil	10L	Local stores	$50
	Bearing shavin	N/A	From work	Nill
Build test rig.	Car engine	2	Dismantlers	$500
	Cooling systen	1	Dismantlers	$100
	Coolant	5L	Local stores	$20
	Fuel	10L	Gas station	$15
	Hardware	N/A	Local stores	$200
	Fabrication	N/A	Help from other	$100
Total estimated cost of resources				$1395

4.6 The Prototype Description and Work Algorithm

The prototype represents an online oil condition monitoring (see Figure 4-15) and vibration analysis system. The main difference between that and inline systems is that online systems do not require direct integration into the tested assembly, whereas inline systems must be placed inside of the oil gallery.

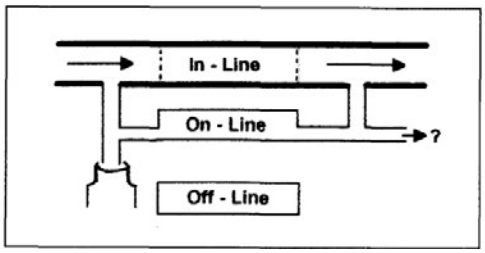


Figure 4-15 - Difference between in-line, on-line and off-line oil sampling (Source :Hunt 1995)

The prototype consists of the following main modules:

1. A gear-type oil pump driven by an electric motor (see Figure 4-16).
2. A filtering element comprising a coarse mesh. The filtering element is placed after the oil pump (see Figure 4-17).
3. Two pressure sensors located before and after the filtering element (see Figure 4-18).
4. One ADXL335 MEMS accelerometer sensor (see Figure 4-19).
5. A signal acquisition and processing module based on the Arduino development board (see Figure 4-20).

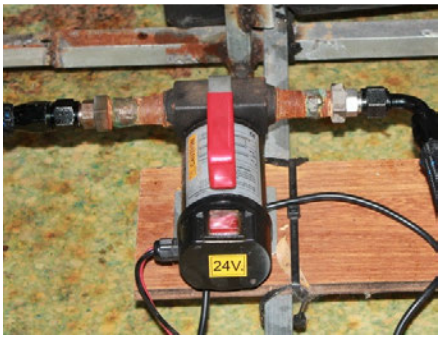


Figure 4-16 - The electric gear pump

Figure 4-17 - The filtering element

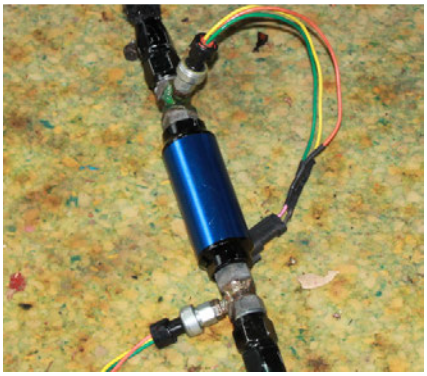


Figure 4-18 - The oil debris collector with pressure sensors.

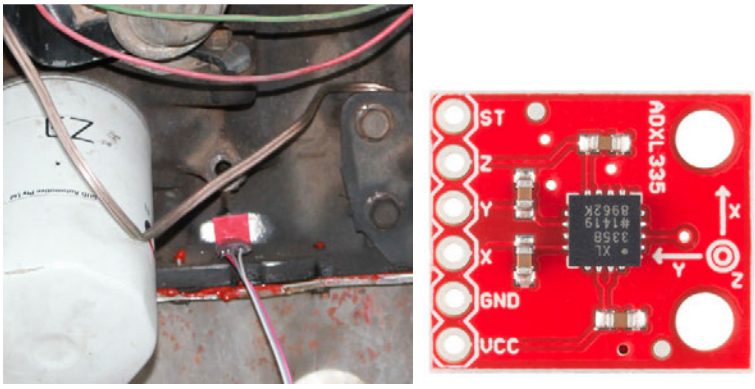


Figure 4-19 - The ADXL335 accelerometer attached to the test engine.

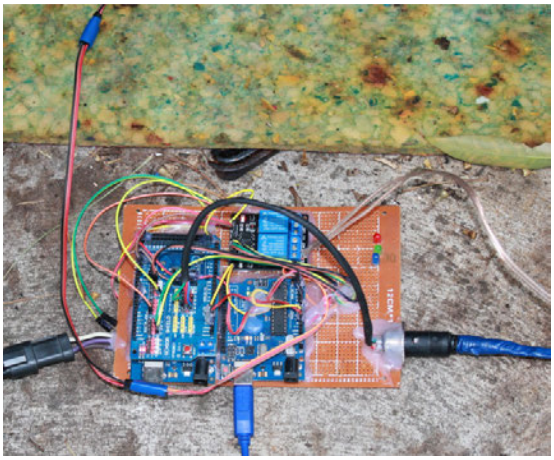


Figure 4-20 - The signal acquisition system (Slave controller (left), Master controller (right), Block of relays (top right))

The prototype's work scheme is as follows. The oil pump maintains the recirculation of the engine oil, collected through a port at the bottom of the engine oil pan (see Figure 4-21) at the flow rate of approximately 20 litres per minute (L/s) (at the oil temperature of 40 degrees Celsius). This flow should be sufficient to detect all contaminants that settle at the bottom of the pan. The oil containing particles flows through the filtering element and is then returned to the pan (see Figure 4-22). Two analog sensors are used to monitor the oil pressure before and after the filtering element.

Figure 4-21 - The oil pick up.

Figure 4-22 - The oil return.

The decision-making algorithm refers to calculated pressure difference read from both pressure sensors (see Appendix E for the C program and the script structure diagram). Blockage of the filtering element caused by large bearing particles restricts the flow, resulting in a rise in pressure in the supply line. The high pressure triggers the alarm signal, indicating engine failure.

The system also monitors the vibration signal generated by the engine. The Arduino board receives the signal from the accelerometer and then sends it to the Matlab program for processing. The Matlab script makes a decision regarding the state of the engine based on the signal analysis. If the script decides that the signal's parameter exceeds the set thresholds, it sends a warning command to the Arduino.

The first stage alarm engages if an excessive knock or vibration is detected. The threshold for the knock/vibration sensor supposed to be established during the rig test (see chapter 8 for further discussions). The first stage alarm is also active if partial filter blockage has occurred. The second stage alarm is triggered if both positive inputs are received or complete filter blockage has occurred (see Table 4-2).

Table 4-2 - Alarm triggering logic.

Vibration or knock	Partial filter blockage	Complete filter blockage
First Stage	First stage	Second stage
Second stage		
Second stage		

The first stage will initiate a warning signal, while the second stage will stop the test engine.

4.7 Test Rig Specification

The test rig comprises a passenger car engine with all supporting subsystems, such as fuel, cooling and exhaust systems. The engine and other components are mounted on a metal frame fabricated from available materials. The frame provides sufficient support for all the elements while functioning (see figures 4-23 and 4-24).

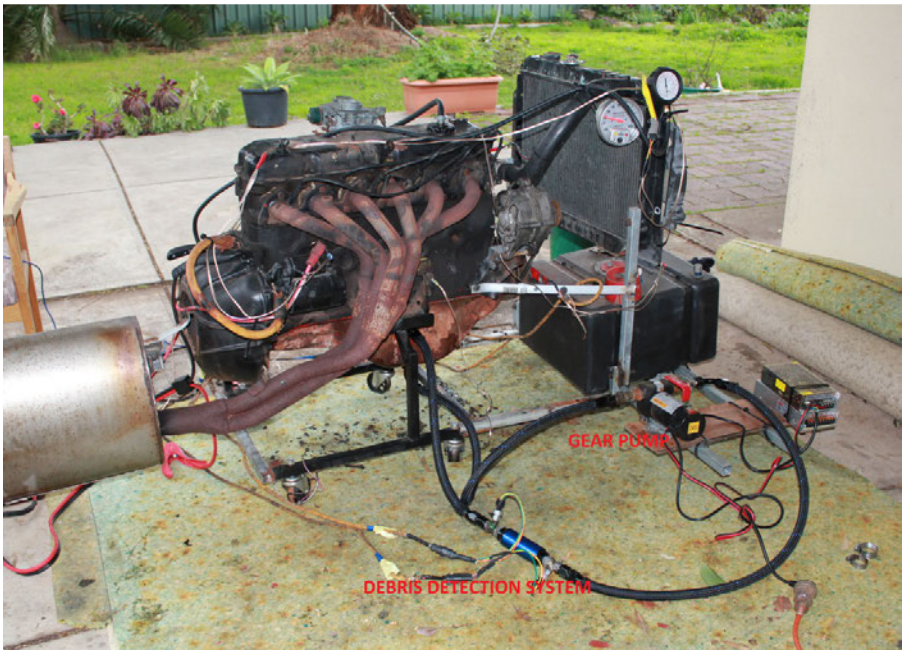


Figure 4-23 - The test rig (R/H view).

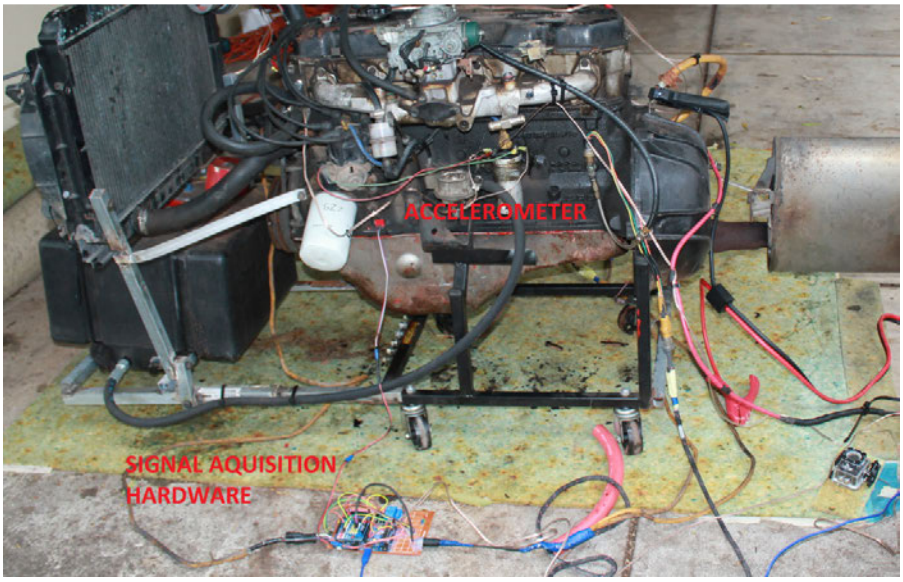


Figure 4-24 - The test rig (L/H view).

4.8 The Testing Procedure

The test itself is subdivided into three stages. Each stage was videotaped and all necessary parameters were controlled and documented. The controlled parameters were the engine oil pressure, the oil temperature, the vibration signal, and the RPM.

The tests procedure was as follows:

1. Firstly, the debris detection system’s performance was evaluated by manually introducing wear debris (see Chapter 5)
2. Secondly, the standard engine running condition was monitored to determine default vibrations and noise levels. The performance of the vibration acquisition system and the optimal location for the accelerometer sensor were also evaluated (see Chapter 6).
3. After the initial runs, the fault condition was introduced by blocking an oil passage in one of the crankshaft's journals. The purpose of this test was to determine the thresholds that would indicate the fault conditions inside of the engine (see Chapter 7). After that, the engine was partially disassembled and the aftermath damage documented (see Chapter 8).

4.9 Risk Assessment

Several of the project stages involved working with power tools, welding equipment and running machinery. Evaluating risks and hazards during each task and developing preventative measures was paramount. The risk assessment matrix (Table 4-3) was adapted from The Australian Centre for Healthcare Governance (2016). Each of the project's tasks involving exposure to risk factors was assessed based on the likelihood of occurrence and potential severity of the impact on a person's health.

Table 4-3 - Risk assessment matrix. (Source: The Australian Centre for Healthcare Governance 2016)

	CONSEQUENCE				
LIKELIHOOD	Insignificant (1)	Minor (2)	Moderate (3)	Major (4)	Extreme (5)
Rare (1)	Low	Low	Low	Low	Low
Unlikely (2)	Low	Low	Low	Medium	Medium
Possible (3)	Low	Low	Medium	Medium	Medium
Likely (4)	Low	Medium	Medium	High	High
Almost certain (5)	Low	Medium	Medium	High	Extreme

Table 4-4 represents the risk-associated tasks and applied control measures. By applying preventative controls and using the necessary safety equipment, it was possible to reduce the likelihood of a risk event taking place and minimise the potential severity of the consequences should one occur.

Table 4-4 - Risk assessment and preventative measures.

Task	Risks level	Level	Preventative actions	Risk level after correction
Build the prototype Using power tools. Cutting, grinding, and drilling.	Eye injury. Lacerations s and broses	Medium	Wear safety boots and glasses all the time. Wear gloves working with power tools and sharp objects. Keep working area free of tripping hazards.	Low
Build the test rig. Using power tools and welding equipment. Cutting, grinding, and drilling. Moderate to heavy lifting.	Eye injury. Lacerations s and broses. Skin burns. Back sprain	Medium	Wear safety boots and glasses all the time. Wear gloves working with power tools and sharp objects. Keep working area free of tripping hazards. Use proper welding gear and a welding mask. Use special lifting equipment or ask for help.	Low
Conduct tests . Running equipment. Hot and flammable liquid escape. Hot gases escape.	Eye injury. Lacerations and broses. Skin burns. Open flame.	Medium	Wear safety boots and a face shield all the time. Wear gloves working with hot objects. Keep working area free of tripping hazards. Stay away of running equipment. Maintain safety barrier before running equipment. Keep fire extinguisher available and ready.	Low

4.10 Quality Assurance

The accuracy of the running tests conducted on the test rig was dependent on several factors. Factors such as engine oil grade, RPM, oil temperature and coolant temperature may all contribute to the significance of the damage to be examined after the initiated fault.

To ensure that accuracy of the test results was not affected by those factors, all experiments were conducted using the same values for the listed parameters. The following values were consistent in each test:

- Engine RPM = 1900-2100
- Oil temperature = 80 degrees Celsius
- Coolant temperature = 80 degrees Celsius
- Oil grade = SAE5W30

Chapter 5 Oil Debris Detection System Test

5.1 Introduction

A number of experiments were conducted to validate the performance characteristics of the oil contamination detection system. The experiments were designed to evaluate parameters that might influence the oil filtering element blockage detection. Two parameters were evaluated: the oil flow rate and the oil temperature. The first parameter affects the time required for contaminants to be trapped by the stream of oil and transferred to the filtering element. The second parameter can potentially cause high differential pressure across the mesh.

5.2 Pre-test assumptions and the size of the filtering element

The size of the test engine is relatively small compared to large industrial applications. At the failure condition, it was assumed that the crankshaft bearings would produce a limited number of metal particles; hence, the filtering element has a small surface area to decrease the time and number of particles needed to block it. The mesh calibre is 250 microns, which represents the average size of wear particles generated during failure mode (see Chapter 2).

Figure 5-1 - The inline filtering element 250 microns.

The filtering element is comprised of a funnel shaped cone that can be inserted inside of a male AN type fitting, and clamped by a female fitting.

5.3 Test prototype setting up

The test prototype consists of the engine sump with the suction outlet located at the lowest point and the return line connected to the side wall. The 24-volt electric oil pump collects the oil from the sump and transfers it through the oil pressure sensor and filtering element (see section 4.5 for figures). Despite the fact that there are two pressure sensors installed, the system reads only one sensor. The second sensor performs a backup function.

It is worth noting that reading the oil pressure in conventional units, such as kPa or PSI, is not of interest. It is only necessary to know whether the reading is low or high; therefore, the sensor is not calibrated to read units of pressure. It reads an analog value resolved in 1024 units, being the range of the ten-bit ADC of the Arduino board. When the system is not running, there is no pressure; hence, the sensor reads zero. When the filter is free of contaminants, the oil pressure sensor would display a small number due to restrictions caused by the mesh and oil lines. When wear particles start blocking the mesh, the pressure reading would start to increase due to restriction.

5.4 Testing procedures

The testing procedure comprises three series of experiments. The first batch of the tests was conducted to evaluate the flow rate and restriction due to the oil lines and mesh. At this stage, clean engine oil, SAE10W40, was used without the addition of contaminants. The testing procedure was performed under the following conditions:

- Cold oil (approximately 12 to 20 degrees Celsius) without filtering element to see the pressure caused by the restriction in the oil lines.
- Cold oil with mesh installed to see the pressure caused by the mesh restriction.
- Warm oil (approximately 40 degrees Celsius) without filtering element.
- Warm oil with mesh installed.

The test results can be seen in tables 5-1 and 5-2. As shown, there is a significant difference in value between the pressure in an open line and when the line is blocked. In addition, the flow rate increases fourfold as the temperature difference reaches almost thirty degrees, which explains the free-flow and the full-block pressure rise. Ideally, the experiment should have utilised a constant flow oil pump; however, this would have significantly increased the project cost. Nevertheless, it is apparent that the filtering mesh of 250 microns introduces the same restriction of 70 units for both cold and warm oils when it is installed in the line.

The second series of tests show whether the system can capture the wear particles dropped in the oil, and how the oil temperature affects the time required to block the mesh. The experiment utilised a coarse aluminium powder to simulate crankshaft bearing wear particles (see Figure 5-2). The powder was dispersed by hand while the pump was running to simulate the wear particles being dropped as the bearing fails. The time lapse between dispersion and the full blockage of the mesh was measured and compared for each test. Whenever mesh blockage occurred, the system was stopped, the oil changed, and the filter washed.

Table 5-1 - The oil flow rate dependence on the oil temperature.

TEST	Flow Rate
Cold oil (12 degrees)	0.08 L/sec
Warm oil (40 degrees)	0.3 L/sec

Table 5-2 - The outcomes of the line restriction tests.

TEST	Free Flow Pressure	Full block pressure
Cold oil no mesh (20 degrees)	70	420
Cold oil with mesh (20 degrees)	140	420
Warm oil no mesh (40 degrees)	100	530
Warm oil with mesh (40 degrees)	170	530

Figure 5-2 - The coarse aluminum powder (left) and the blocked filtering mesh (right).

5.5 Comparison of the Test Runs with Cold and Warm Oil

Figure 5-3 shows the timeframe from introduction of the particles to the moment when the pressure reaches full blockage pressure. It is evident that the time required for the particles to block the filter is almost identical for both cases. As shown, it takes approximately ten seconds for the debris to start reaching the mesh. Over the next 13 seconds there is a gradual rise in pressure until the full blockage condition occurs at the 25th second.

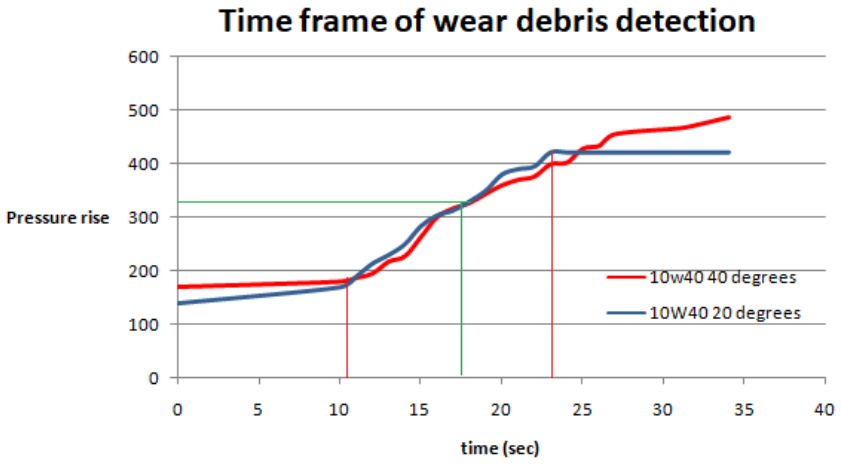


Figure 5-3 - Time required by the system to detect the debris (Produced with the aid of MATLB 2016A student version).

It is possible to conclude that it is unnecessary to wait until the pressure reading reaches full blockage. The threshold can be set to the middle of the inclined line, which represents about 320 of the pressure units. This chosen threshold adjusts the time required by the system to detect wear debris to 18 seconds.

It is worth noting that the experiments did not take into account the flow that can be created by the engine oil suction line. If the failing bearing is located in the proximity of the oil pick-up, the generated debris may be trapped by the engine oil system, failing

to reach the prototype system. For this project, the oil pick-up was diverted away from the test system inlet.

5.6 Conclusions

A series of experiments were conducted to evaluate the time taken to block the filtering element, and test whether the oil temperature and the restriction caused by the 250-micron mesh would cause considerable pressure rise. The tests produced the following outcomes:

- The oil temperature did not significantly affect the pressure rise.
- The restriction due to the mesh size of 250 microns was negligible in comparison with the full blockage pressure.
- The full blockage pressure was considerably higher than the open line pressure, regardless of the oil temperature.
- The time taken to block the filter was independent of the oil temperature.
- The oil flow rate affected the system's ability to capture the wear particles. For this reason, it is recommended to maintain the flow rate at as high a level as possible.

Based on these results, it can be concluded that, for the Australian climate, the restriction imposed by the filtering mesh due to the increased viscosity at low temperatures does not affect the system's performance considerably. The oil pump used in the system, however, was not able to maintain a constant flow rate, regardless of the temperature. It is not known, therefore, whether the same flow rate of cold oil would elevate the pressure rise across the filter due to mesh restriction. Nevertheless, the variation of the flow rate can be regarded as a simulation of the cold oil bypass valve that is usually implemented in the oil coolers or the oil filter housing of contemporary engines.

It is worth noting that if the system is used in sub-zero temperatures, the high viscosity issue must be addressed.

Chapter 6 Vibration Acquisition and Processing System Tests

6.1 Introduction

An evaluation of the performance of the vibration data acquisition and process system was undertaken. This was determined the best location for the accelerometer since it would be able to provide a clean and informative signal.

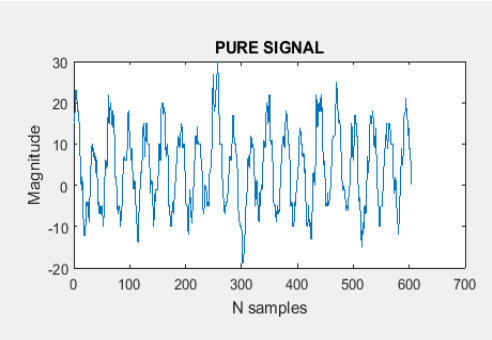
The accelerometer was placed at the piston, top dead centre, at the middle of the stroke and to the axis of the crankshaft. The misfire condition was simulated by disconnecting one or two power leads from the spark plugs at about 1900rpm.The tests revealed that the most compelling and informative signal could be obtained when the location of the accelerometer was on the axis of the crankshaft. From this place, the magnitude of the peaks of the main harmonics was prominent and could be easily distinguished from the noise.

In the following section, the abovementioned findings, as well as other aspects, will be discussed in detail.

6.2 Comparison of the Signal in Time Domain

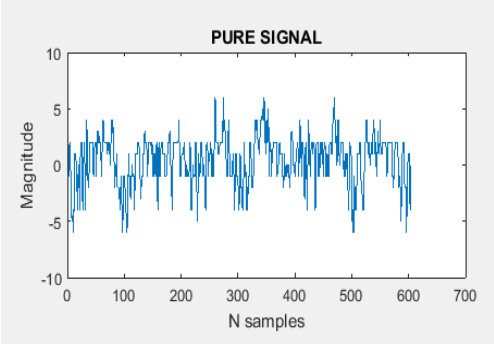
Bottom of the block

1800 RPM



Middle of the block

1800 RPM



Top of the block

1800 RPM

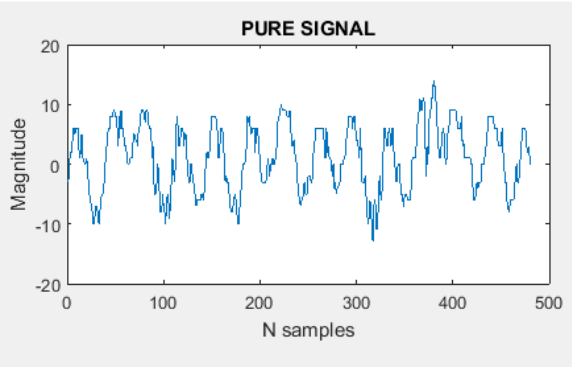


Figure 6-1 - Comparison of raw signal obtained at three different locations at the healthy engine condition (Produced with the aid of MATLB 2016A student version).

Figure 6-1 shows the magnitudes of the raw signal measured at three different locations. As shown, the signal is clearest at the bottom of the cylinder block. At the middle of the block, the signal is weak and contains an excessive amount of noise. The difference in these signals can be explained based on a study conducted by Broatch, et al. (2008). They concluded that an engine could be regarded as a rigid body oscillating about its centre of rotation (see Figure 6-2), where the position of the centre depends on the mounting of the engine. The signal measured by an accelerometer represents the tangential acceleration. The greater the distance from the centre, the greater the magnitude of the acceleration is.

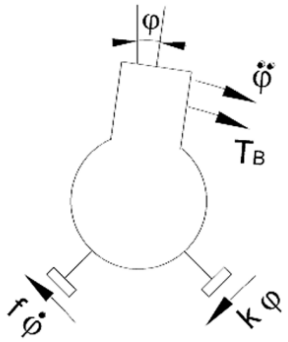


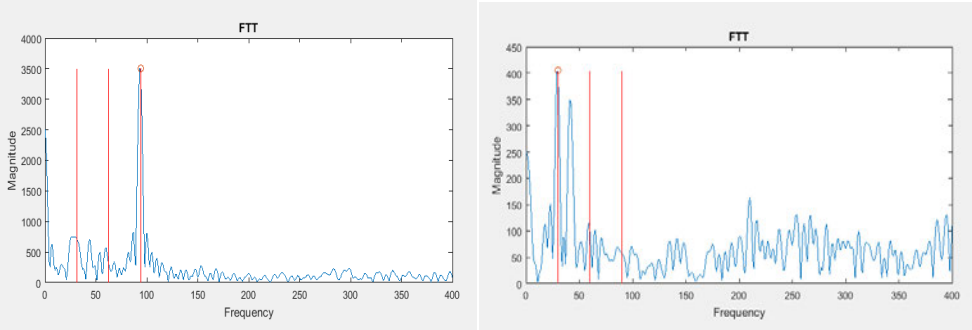
Figure 6-2 - Torsion oscillation of the engine block. ω - angular displacement, ώ - angular velocity, ῶ - angular acceleration, T - torque, f - frequency, k - spring constant (Source : Broach et al. 2008).

It can be concluded, therefore, that the centre of oscillation for the test engine should be located in the middle of the cylinder block, as the position at the bottom of the block represents the farthest point from the centre of oscillation.

6.3 Comparison of the Fast Fourier Decompositions

In Figure 6-3 it can be seen that at the intermediate location, the FFT is contaminated by noise frequencies, making it difficult to determine the main frequencies of the engine, whereas the FFT of the signal obtained at the bottom of the block reveals the main harmonics. Here, the RPM reading was synchronised with the FFT plot. The three vertical red lines correspond to RPM/60, 2*RPM/60, and 3*RPM/60, respectively. This allows correlation of the engine RPM with the first, second and third harmonic orders.

Bottom the block **Middle of the block**



Top of the block

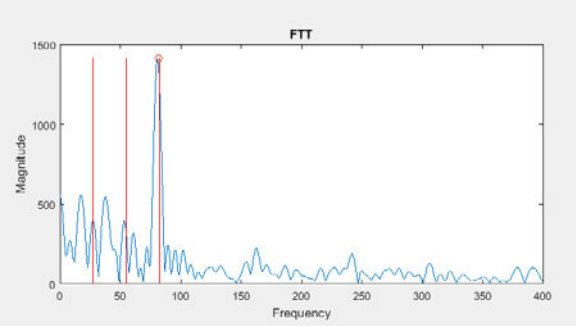


Figure 6-3 - Magnitudes to noise ratio of the FFT of the signal obtained at different locations (Produced with the aid of MATLB 2016A student version).

6.4 Determining the Optimum Location for the Accelerometer

6.4.1 Healthy Conditions

Before taking the vibration measurements, the engine was warmed to the temperature of the coolant, 80 degrees Celsius. The accelerometer was then positioned at the bottom of the block near the second crank pin. At the beginning, the engine was running at approximately 1900rpm. The signal was sent to the Matlab GUI program, which performed data logging, FFT decomposition, calculation of RMS and Crest Factor values, and real-time plotting of the data.

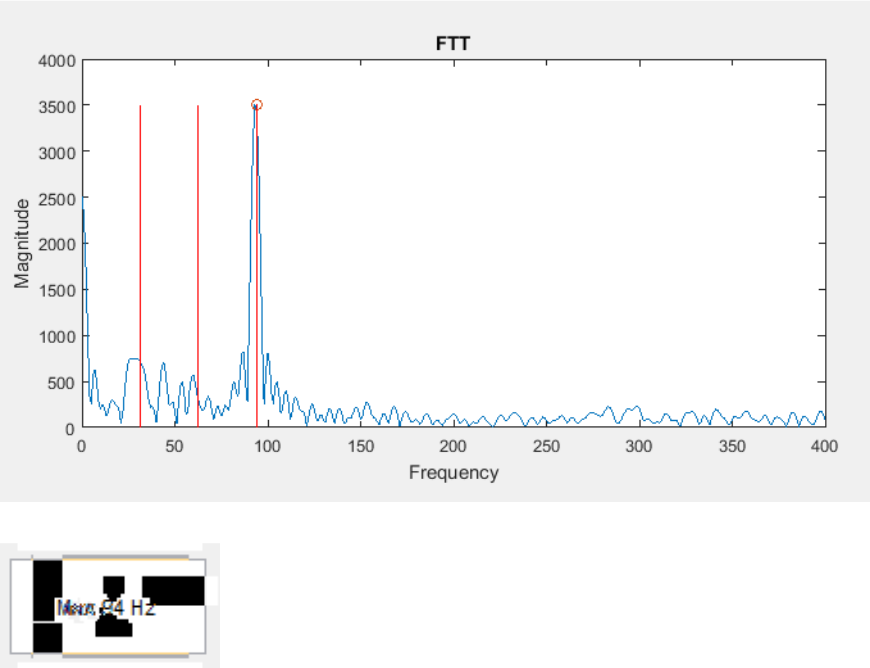


Figure 6-4 - FFT decomposition of the signal at normal condition (1900 RPM) (Produced with the aid of MATLB 2016A student version).

Figure 6-4 represents the vibration signal FFT decomposition. The vertical red lines represent the first three harmonic orders. The torque pulses caused by combustion contribute most to the first few harmonic orders. Combustion frequency can be estimated as follows (Li, et al., 2016):

$$f = \frac{2}{60c} ni$$

Where:

n = RPM

c = number of strokes (e.g. two- or four-stroke engine)

i = number of cylinders.

Substituting known parameters into the equation, we obtain:

$$f = \frac{2}{60 * 4} 1900 * 4 = 95\ Hz$$

As such, it is apparent that the third order harmonic is the combustion frequency. For an engine in healthy condition, this dominates over the first two harmonics; however, the first order harmonic, which is the rotational frequency, is not apparent.

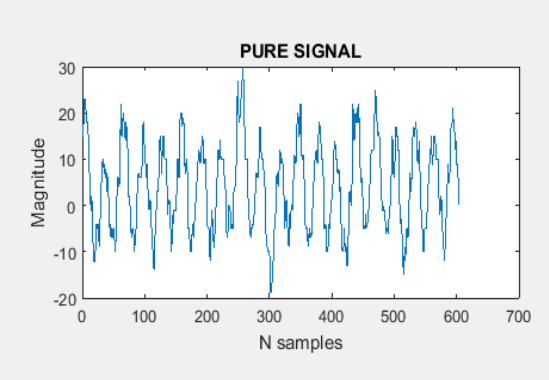


Figure 6-5 - Raw signal of the healthy engine (Produced with the aid of MATLB 2016A student version).

The raw vibration signal from the healthy engine consists of an apparent sinusoidal curve (see Figure 6-5).

6.4.2 One Cylinder Disconnected

Simulation of a fault by disconnecting one of the cylinders produced the following outcomes.

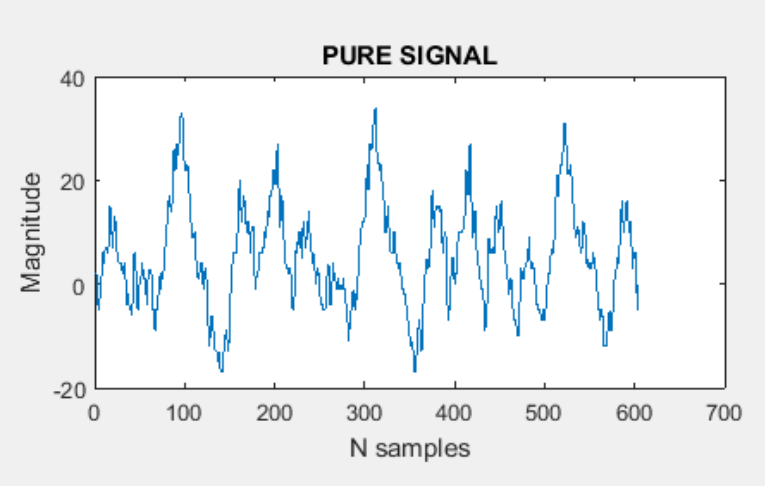
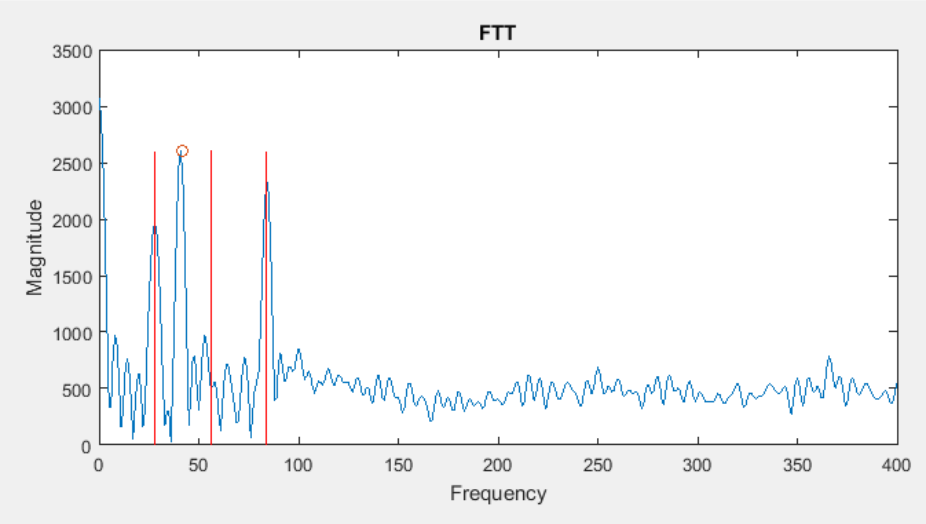


Figure 6-6 - Raw signal of the engine with number six cylinder disconnected (Produced with the aid of MATLB 2016A student version).

When the number six cylinder was disconnected, the engine RPM dropped to 1650. On Figure 6-6, it is evident that the pattern of the raw signal has noticeably altered compared with that from the healthy signal.



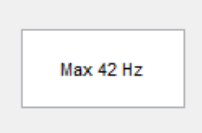


Figure 6-7 - FFT decomposition of the signal at number six cylinder disconnected (1650 RPM) (Produced with the aid of MATLB 2016A student version).

The FFT graph (Figure 6-7) shows that the dominating harmonic has shifted to between the first and second orders. It can also be seen that the first order frequency is more prominent than it was when there was no fault, while the magnitude of the third order harmonic has reduced from 3500 to 2500.

6.4.3 Two Cylinders Disconnected

Figure 6-8 depicts the raw signal obtained when the number six and number five cylinders were disconnected. Again, it is evident that the vibration profile has been significantly affected.

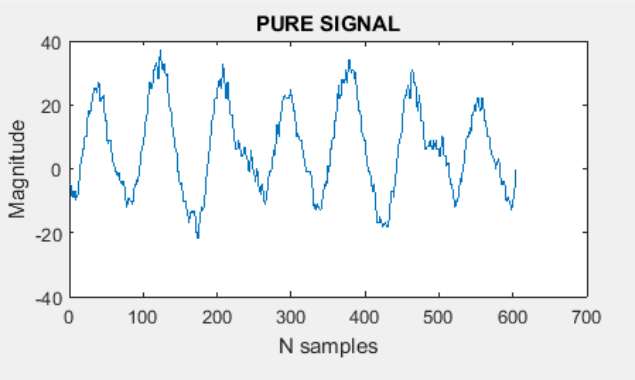
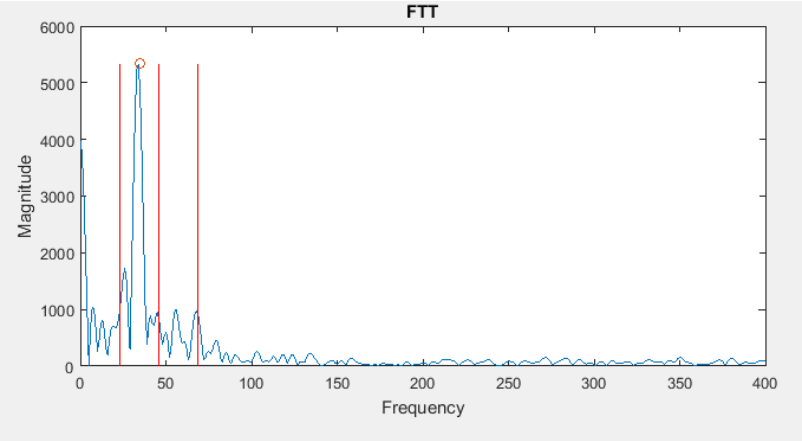


Figure 6-8 - Raw signal of the engine with number six and five cylinders disconnected (Produced with the aid of MATLB 2016A student version).

Figure 6-9 clearly shows that the first and the third harmonics diminished; however, the magnitude of the half order harmonic doubled in comparison to when only one cylinder was disconnected.



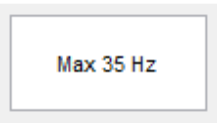


Figure 6-9 - FFT decomposition of the signal at number six cylinder disconnected (1650 RPM) (Produced with the aid of MATLB 2016A student version).

6.4.4 RMS, Crest Factor and Spectrogram

Figures 6-10, 6-11 and 6-12 represent the RMS and Crest Factor plot, and the spectrogram in 2D and 3D, respectively. The data is plotted versus the time taken to perform the whole test run, including the healthy condition stage and the simulated misfiring.

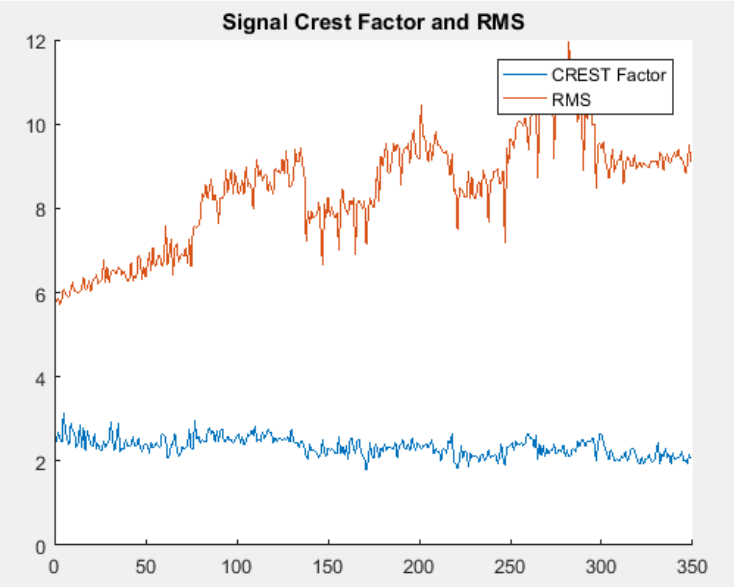


Figure 6-10 - Crest factor an RMS plots (Produced with the aid of MATLB 2016A student version).

From Figure 6-10, it is clear that the RMS value has three sudden rises, which are related to the disconnection and re-engagement of the cylinders. It is evident that the RMS is sensitive to the misfire. The last rise of the RMS depicts where two cylinders were disengaged. In the Crest Factor, however, there seems to be no response to the changes in operation of the cylinders.

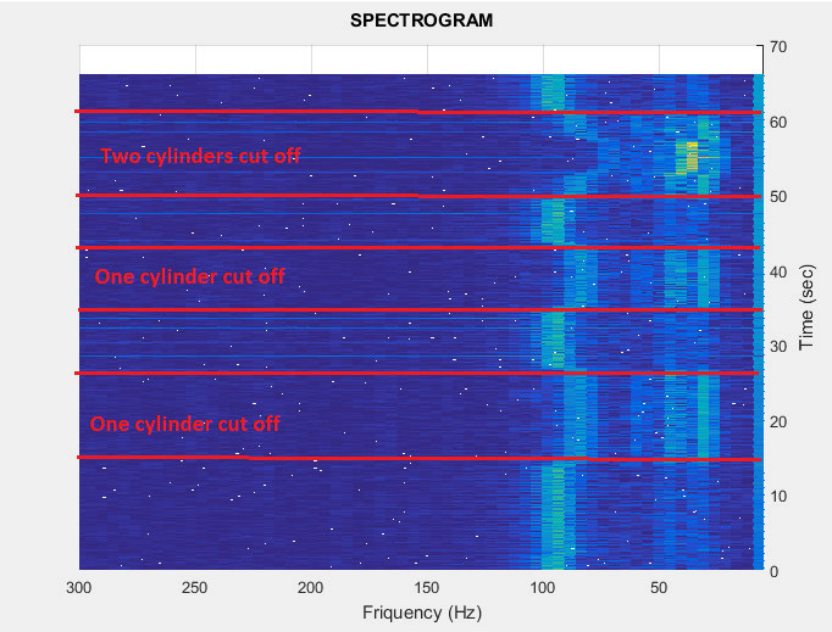


Figure 6-11 - Spectrogram of the signal resolved in time domain (2D) (Produced with the aid of MATLB 2016A student version).

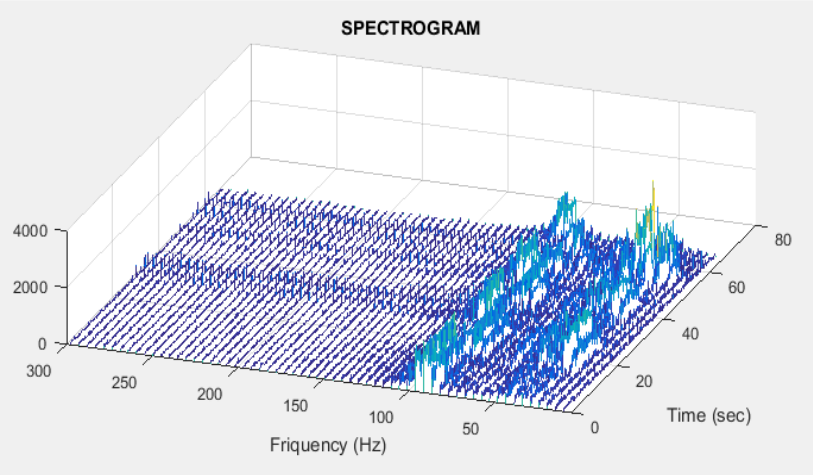


Figure 6-12 - Spectrogram of the signal resolved in time domain (3D) (Produced with the aid of MATLB 2016A student version).

Figures 6-11 and 6-12 provide a clear representation of events occurring during the tests. As shown, the engine ran for sixty-five seconds. Shifts in frequency can be seen at 15, 35 and 50 seconds. With respect to these, the spectrogram plot provides information regarding any drift of the data from the reference parameters.

6.5 Conclusions

The initial set of experiments were conducted to determine the optimal location for the accelerometer and to examine the system's response to a simulated fault, such as a misfire. Furthermore, the signal processing functions, such as RMS, Crest Factor, FFT, and spectrogram were evaluated in terms of their suitability for fault detection.
Comparing the presence of noise, as well as the strength of the signal, the best location for the accelerometer was determined to be close to the axis of the crankshaft. The signal read from this position exhibited sufficient readability and signal/noise ratio.

During the simulation of the misfire, it was noted that the RMS value reacted rapidly to the altered conditions, whereas the Crest Factor failed to provide any detectable response. The FFT plot showed distinct frequency shifts in the main harmonics, while the spectrogram plot provided a clear representation of the frequency change when a misfire was introduced. Despite the possible application of the spectrogram to examine the logged data, however, it cannot be used during online monitoring.

Chapter 7 Determining the Failure Thresholds

7.1 Introduction

The experiments were conducted to understand the processes occurring at the initiation and aggravation of a failure in the engine, and to evaluate the systems' performance. Before the trial run, the cooling system was warmed to 50 degrees Celsius, and the engine oil temperature increased to 70 degrees Celsius. At this oil temperature, the oil pump used in the debris detection system provided 0.5 L/s of flow; therefore, the entire volume of engine oil was circulated through the filtering mesh every ten seconds.

The failure mode was initiated by blocking the oil supply hole at the #2 big end bearing (see Figure 7-1). Initially, it was assumed that simulating oil starvation would be sufficient to instigate a failure; however, after the first twenty minutes, there was no noticeable change in the engine's operation. Neither a change in the vibration profile was detected, nor did the full blockage of the debris detection system occur. Subsequently, the engine was stopped and the bearing removed for examination. No signs of severe damage to the bearing were found.

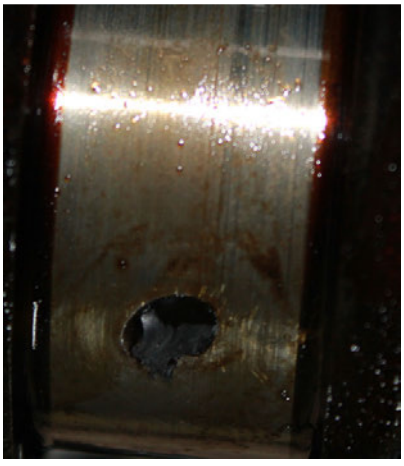


Figure 7-1 - A piece of rubber blocks the oil supply hole.

Based on experience, it was concluded that the main drawback of the current test set-up was that the engine could not be loaded by opposing torque; therefore, marginal lubrication due to oil splashing provided sufficient lubricant to the bearing to maintain its stable state. The decision was made to reduce the bearing/journal clearance and repeat the test. The clearance was reduced by placing a couple of strips of aluminium foil under the bearing of the connecting rod (conrod) cap (see Figure 7-2). After the tensioning of the cap, there was a noticeable resistance to the engine rotation.

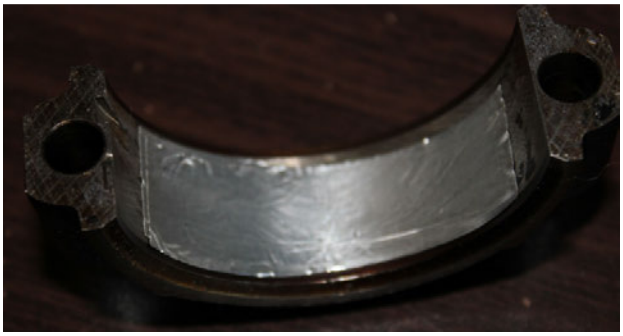


Figure 7-2 - Aluminum foil was placed at the conrod cap to reduce the oil clearance.

After the first five minutes of the second run, a noticeable ticking noise became apparent; however, the pressure rise in the debris detection system indicated only partial blockage of the filter. For this reason, the engine was left to operate for another 20 minutes to investigate whether the failure would worsen; however, no obvious degradation of the running condition was detected. The pressure rise due to the debris being trapped by the filtering mesh reached only half the maximum value.

7.2 Discussion on Debris Formation and the Oil Clearance

A second examination of the bearings revealed that the top bearing underlay had vanished and the copper lining appeared visible. Nevertheless, the test had failed to achieve severe damage to the crankshaft bearing. One possible explanation for this phenomenon is that the top layer of the bearing had worn down, allowing sufficient oil clearance to consequently restore marginal lubrication of the journal.

The processes at work inside of the engine during the second run can be outlined in the following sequence:

1. In the beginning, due to reduced oil clearance, the temperature at the bearing journal interface rose rapidly, forcing the top bearing layer to flake off or melt. At this stage, the rate of wear debris generation was high enough to partially block the filtering mesh.
2. Rapid wear of the bearing's top overlay provided sufficient oil clearance, which restored marginal lubrication of the journal. The temperature of the journal decreased, and the rate of wear particles generation dropped to a minimum, which was insufficient to trigger the debris detection system.

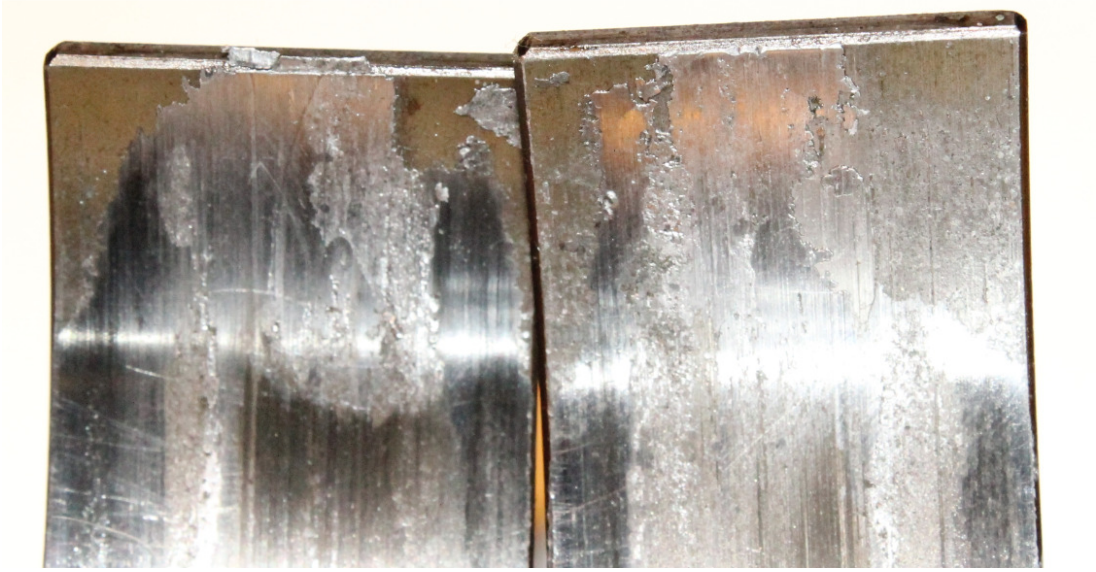


Figure 7-3 - Melting and smearing of the bearing's top overlay.

Figure 7-4 - Molten droplets of the failing bearing's top overlay being captured by the debris detection system.

As shown in figure 7-3, the tin lining started to melt and became smeared over the bearing while the molten droplets of tin were pushed outside into the oil. The pictures of the filtering mesh (see Figure 7-4) show the captured molten droplets.

Unfortunately, without loading the engine, it was impossible to initiate rapid and severe bearing degradation that would produce enough debris to clog the mesh completely.

7.3 Discussion Regarding the Frequency Analysis

During the test run, vibration data was recorded and analysed using the replay feature of the Matlab GUI program.

Figure 7-5 shows plots of the FFT and the pure signal waveform. The Kalman filter was applied to provide effective noise reduction of the signal. It is clear that the dominating frequency is the third harmonic, induced by fuel combustion.

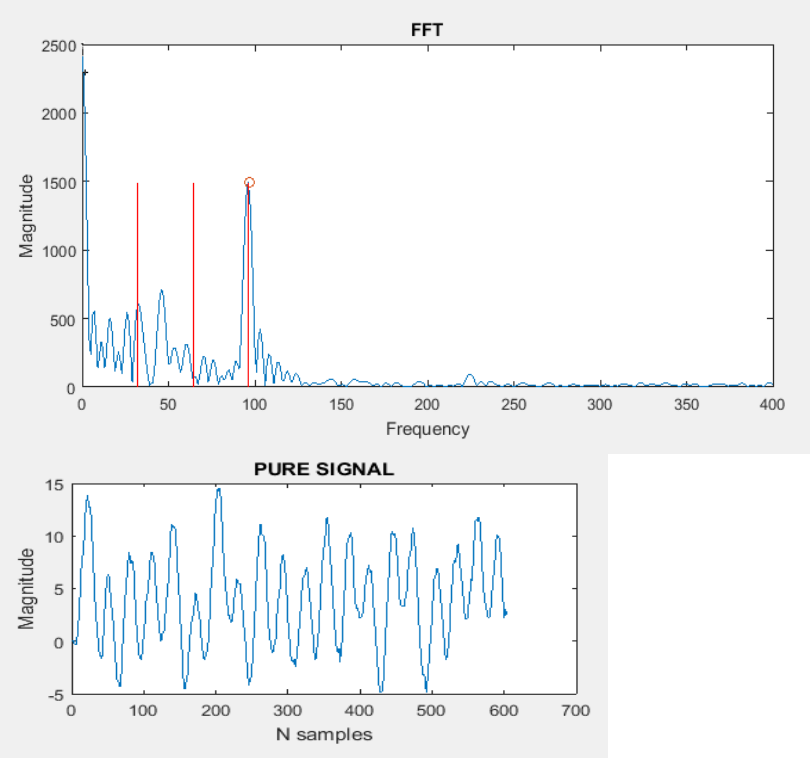


Figure 7-5- Frequency and pure signal profiles before the failure at 1900 RPM (Produced with the aid of MATLB 2016A student version).

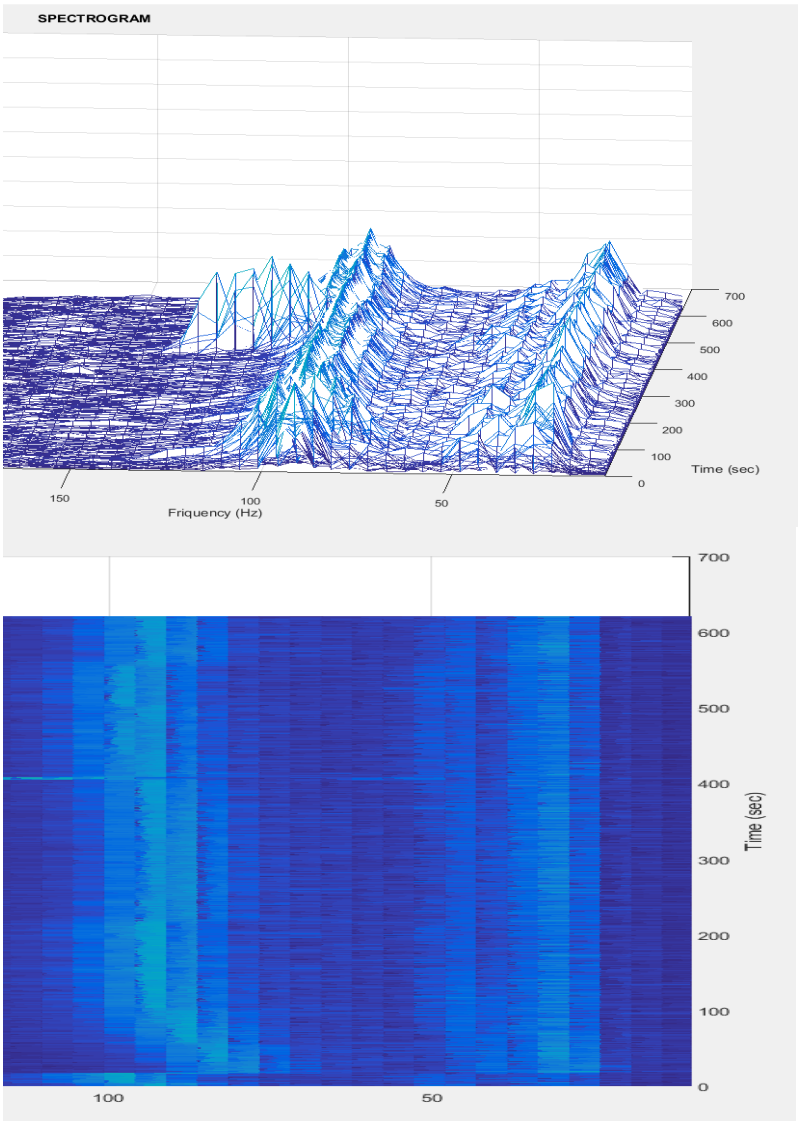
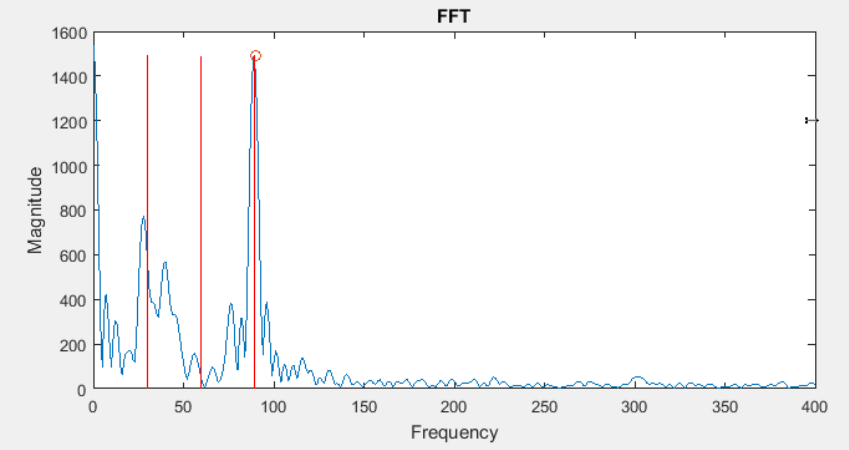


Figure 7-6 - Spectrogram before the failure initiation at 1900 RPM (Produced with the aid of MATLB 2016A student version).

The spectrogram shown in Figure 7-6 distinguishes the first and the third harmonics associated with 1900rpm (light blue regions). No apparent spikes or sudden frequency shifts can be seen.



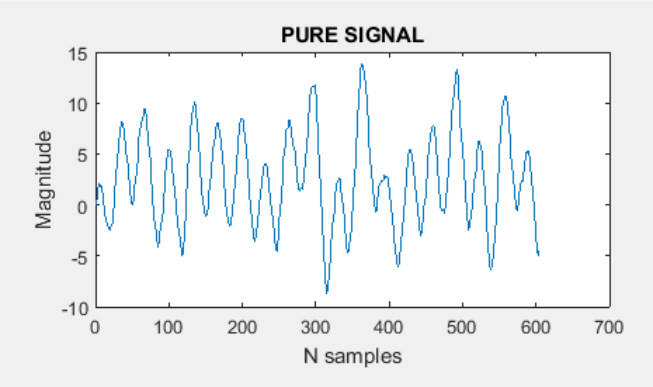
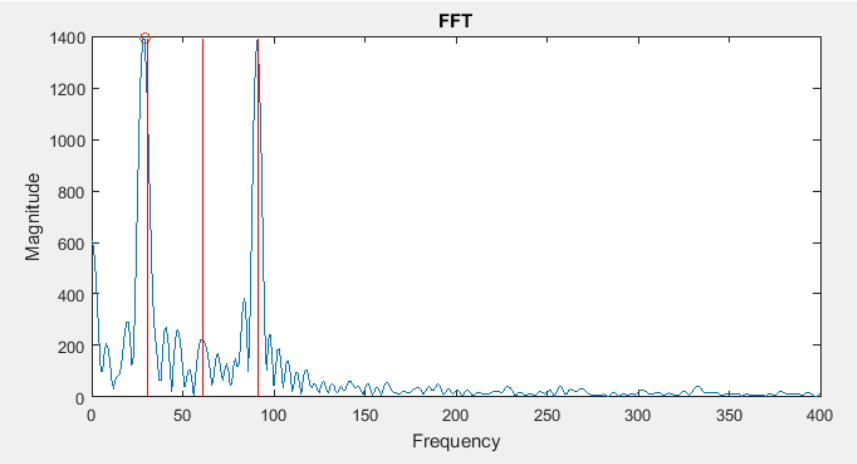


Figure 7-7 - Beginning of the failure at 1900 RPM (Produced with the aid of MATLB 2016A student version).

As the failure began to aggravate, the first order harmonic became amplified, as evident in Figure 7-7. Nevertheless, the signal waveform was apparently unaltered.

Figure 7-8 shows that, as the failure progressed, the third harmonic became overwhelmed by the rotation frequency. Moreover, the waveform profile altered substantially.



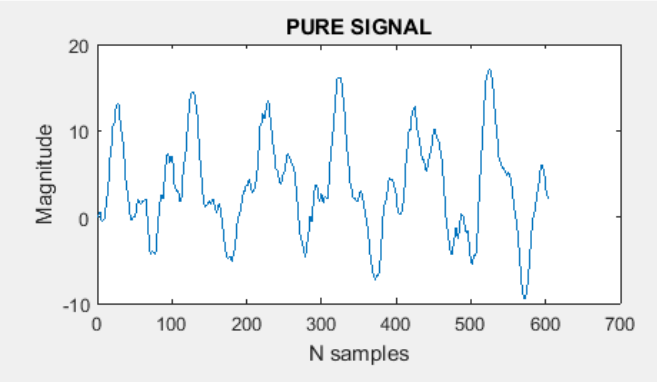


Figure 7-8 - Aggravation of the failure at 1900 RPM (Produced with the aid of MATLB 2016A student version).

Figure 7-9 represents the frequency spectrogram of the vibration data collected during the first eight minutes of the second run. On the plot, there are three distinct regions. The first region reflects the standard running condition of the first two minutes of the test run. During this period, it was expected that the top bearing overlay would provide the sliding surface with a low friction coefficient. At this stage, it is evident that the dominating frequency is 100Hz, which is the firing frequency. The rotation frequency of 33Hz is also apparent.

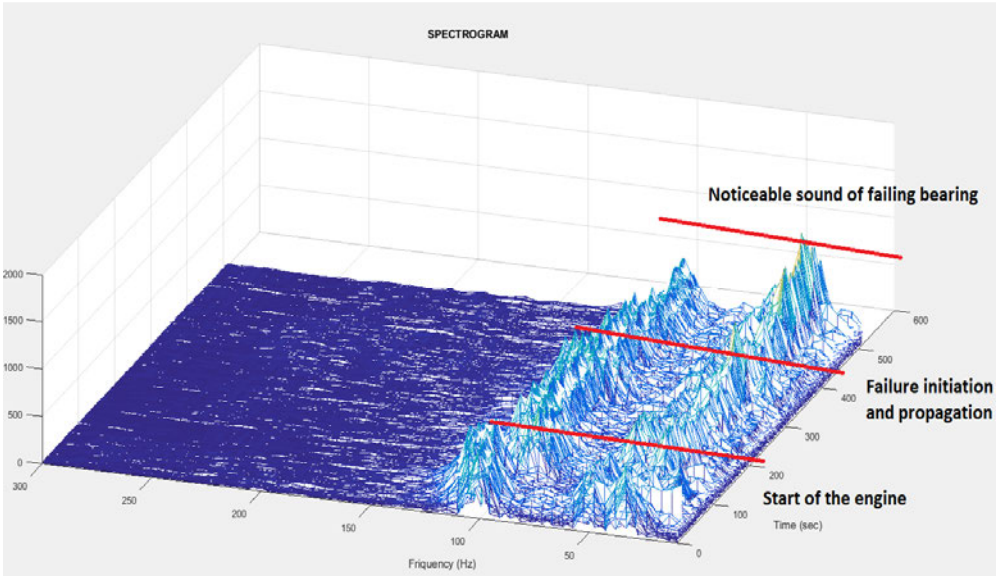


Figure 7-9 - Spectrogram of the test engine run with induced failure at 1900 RPM (Produced with the aid of MATLB 2016A student version).

With marginal lubrication, however, the temperature rise due to friction would cause the top layer to melt away, exposing the second bearing overlay. The second overlay is made of copper, which has a relatively higher friction coefficient than the top overlay; therefore, it is assumed that the second region of the spectrogram represents the condition of the top layer starting to break away.

The third region reveals that the rotation frequency started gradually increasing to the point where it was dominating other frequencies indicating abnormalities in the engine condition.

7.4 Discussion Regarding the RMS and the Crest Factor Values

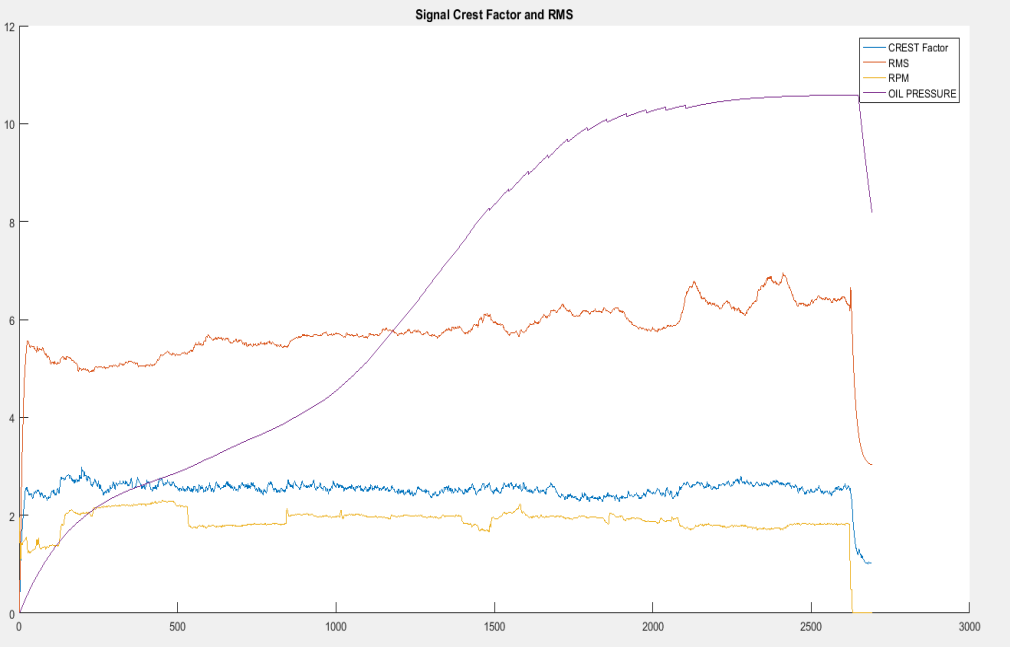


Figure 7-10 - The plot of the history changes of RMS, the crest factor, RPM and the oil pressure rise across the debris detection system. (Produced with the aid of MATLB 2016A student version, see Appendix F for bigger picture).

Figure 7-10 plots the Crest Factor (blue line) and RMS (orange line) calculated over the course of the test run. Two additional lines were inserted to visually represent the correlation between the RPM readings (yellow line) and the pressure rise across the debris detection system (purple line) and the other two readings. The RPM and pressure lines were scaled to fit into the plotting space. The function of the plot is to represent the correlation between these four parameters; hence, it is unnecessary to provide the exact values of RPM and oil pressure rise. Showing how the parameters are interconnected is sufficient. In the previous experiments (see Chapter 6), it was demonstrated that the Crest Factor and the RMS were reliable indicators of changes in the vibration profile due to misfire.

Figure 7-10 demonstrates that the RPM remains relatively stable. The pressure curve exhibits a steady rise, followed by a plateau region. The plateau may indicate the moment when the oil clearance between the crankshaft journal and the bearing was restored; thus, the debris formation process had ceased. The RMS curve clearly shows sudden peaks in the last quarter of the plot. This may be an indication of a temporary bearing seizure, which generates an opposing momentum to rotation resulting in fluctuation of the torque delivered by the engine (Broatch, et al., 2008).

The Crest Factor remained stable throughout the entire run. Accelerometer error due to the engine temperature rise may explain the slight drift of the RMS data towards the top. It can be concluded that, while the Crest Factor may provide a suitable response to a gradually propagating failure, it has limited application where rapidly progressing malfunctions occur.

7.5 Conclusions

The set of experiments including an induced failure mode revealed that there is potential for the system to perform as expected during the design phase. At this stage, the experiment was partially successful. The main limitation of the test rig was the absence of the loading. As such, the experiment failed to initiate rapid or catastrophic failures of the crankshaft bearings to test the full potential of the proposed system. Nevertheless, the debris detection system managed to capture some of the wear particles. Moreover, it was shown that changes in the vibration data could be detected as the failure propagated.

The main disadvantage of the debris detection system was revealed, however. Since the pick-up of the engine oil system is located in proximity to the debris detection intake, it is possible that the majority of wear particles were captured by the engine oil filter. Even diverting the pick-up tube away from the system's oil collector did not improve the situation. In order to avoid this, the debris detection system should be incorporated into the engine oil circuit. The engine oil pump would ingest the oil from the sump, and

transfer it through the debris detector before passing it through the main oil filter. A bypass valve should also be included to prevent a complete cut in oil supply due to filtering mesh blockage.

Chapter 8 Discussions and Conclusions

8.1 Introduction

The primary objective of this project to research, design and assess the proposed system was fulfilled with partial success. The initial test of the system produced positive responses. The debris detection system was able to react promptly to manually dispersed metal particles while the vibration processing system reacted to simulated misfire. The experiments performed on the test engine had failed to achieve the rapid bearing degradation or delaminating. As stated previously, it was impractical to sufficiently load the test engine without a dynamometer. For this reason, the full potential of the proposed system could not be evaluated. Although the system showed some positive response to the partially failing bearing, further experiments with a properly loaded engine are required.

8.2 Discussions

8.2.1 The Debris Detection System

In the early stages of the project, potential problems associated with filtering mesh restriction were addressed. Tests were conducted using two different temperatures of the test oil: ten and 40 degrees Celsius. This set of tests revealed that the mesh used in the system causes minimal restriction compared to the pressure rise induced by full blockage. It was found, however, that the oil temperature has a negligible effect on the flow restriction.

It is worth repeating that these results are only valid for this particular arrangement. First of all, the system's oil pump is unable to maintain a stable flow rate. The clearances inside of the pump's housing allow the oil to be regurgitated if full blockage of the mesh occurs. Pumping the colder oil required increased power input, which slowed down the pump, equalising the power to flow rate ratio. Utilising this particular pump eliminated the need for a bypass valve.

It was suggested that high oil viscosity might affect machinery operating in cold climates. The bypass valve must be included in cases where constant oil flow is provided. The valve will ensure that, in the event of filter blockage or a high rise in pressure due to viscosity, ruptures to the supply lines do not occur. The bypass valve can act as a switch to trigger the warning. In this case, it is advised that the system controller should be programmed to ignore the opened valve during a cold start or operate the pump only when the oil is warm.

Another aspect of the debris detection system requires particular attention. The location of the intake line of the system is fixed, whereas any of the crankshaft bearings may fail. As the length of large industrial engines may exceed two metres, aids to improve the debris pick-up rate should be discussed. One suggestion could be to use a number of intake ports, equally spaced along the oil sump. Moreover, funnels could be utilised to guide the particles into the ports.

One more issue with the test engine that potentially had an effect on the debris detection system was that the pick-up tube of the engine oil pump was situated close to the intake of the debris detection system. It is assumed that the engine oil filter will capture the majority of wear particles. One possible solution, therefore, could be to insert the debris detection system before the engine oil pump. In this way, the engine oil pump would be transferring the oil from the sump through the debris detector before passing it to the main oil filter. The design must incorporate a bypass valve to prevent oil supply interruption if the filtering element becomes fully blocked.

The abovementioned solution significantly complicates the system, compromising one of the design goals, which was to design a system that is easy to implement. The majority of industrial engines have several holes already drilled and plugged in the sump to connect various accessories; hence, multiple intake points would be the most viable solution, as they could utilise existing holes.

8.2.2 The Vibration Acquisition Hardware

The experiments showed that the current hardware set-up is capable of delivering adequate performance to comply with the design requirements. Moreover, the sampling rate should be sufficient to avoid the aliasing effect. There is, however, a bottleneck between the Arduino ADC and the serial port due to the low frequency of the main buss. The bottleneck creates gaps between the data block because the controller does not read the new information while processing the buffer. It is possible that certain events can be overlooked if they occur during these gaps. The author discovered that the STM32 microcontroller has a much greater serial transfer speed than that of Arduino. The STM32 controller is compatible with the Arduino IDE and can be used to improve the system. Despite this fact, there was not sufficient time to adapt the existing design to employ the STM32.

Another aspect of the hardware is the deficiencies of the ADXL335 accelerometer. Tests revealed that it is sensitive to temperature rise, which induces a drift of the data from the zero reference. Moreover, the limited bandwidth and the excessive noise affect the accuracy of the readings significantly; therefore, it is advisable to implement industrial standard accelerometers from manufacturers such as ENDEVCO or Brüel and Kjaer. These are piezoelectric types of accelerometers, however, require additional amplification circuitry.

8.2.3 The Vibration Processing

Noise was consistently present in the data and became one of the main issues that challenged the project in the normalisation of erratic readings. It appears that the developed Kalman filter provides the most efficient and adequate smoothing of parameters obtained from the sensors. It has been found that the data should be processed by the Kalman filter twice to reveal reliable trends in the RMS plot. This approach is used for offline datalog analysis. Firstly, the raw vibration data is passed through the Kalman filter before a special function calculates the RMS value of the

filtered data. Finally, the array of RMS values obtained during the experiment passes through the Kalman filter a second time. The obtained curve provides the most precise correlation between the RMS and other parameters.

In real-time signal processing, the main Matlab script can be improved to perform in the same manner as the offline processing script. Normally, the system receives raw signals in blocks, which are passed through the chosen filter. The RMS and other parameters are then extracted and plotted immediately. An improved algorithm should combine a user-defined number of RMS readings into the array, to be passed through the Kalman filter again before plotting. The length of the array should be determined experimentally.

8.2.4 The GUI Interface

The GUI interface evolved into a standalone tool that can be used outside of the proposed system. It could be employed to analyse any vibration data saved in *.csv format. The current version of GUI contains all necessary signal processing functions and filters, such as the spectrogram, RMS, and Crest Factor. The outputs provided by the GUI could then be used to analyse the signal, revealing its features and trends effectively.

8.2.5 Safety Considerations Associated with Implementation of the System

One of the concerns with the commissioning of the proposed system is that it could potentially induce a probability of failure. The failure would mainly occur due to a rupture in one of the lines of the debris detection system. A suggested solution is to use high quality steel braided lines, as the maximum burst pressure of steel braided lines significantly exceeds pressure in the system. The lines would need to be sufficiently

secured with proper mounting hardware and located away from the exhaust manifolds. If there is a risk of the engine oil coming into contact with the hot exhaust, the lines must be shielded.

8.3 Further Work

As outlined in the above paragraphs, there is huge potential for improving the system. The performance and accuracy of the signal acquisition hardware could be enhanced by implementing an STM32® microcontroller and an industrial accelerometer. It is also possible to compile the Matlab script into a standalone application to be embedded in a microcomputer such as Raspberry Pi, which would allow the development of a black box solution. Online processing of the RMS value could be improved by two-stage filtering, which it is already possible to implement in offline processing; however, the online version will require additional buffering of the RMS data.

Issues concerning the debris detection system can be addressed by reviewing the design regarding the number of oil intake ports and designing the housing to accommodate a bypass valve.

Finally, the most important amendment will be to conduct future experiments with the test engine mounted on a dynamometer stand to enable application of a full load.

8.4 Conclusions

The project culminated in the design, development and testing of an easily implementable and straightforward system to prevent the catastrophic failures of industrial engines. The system is able to detect the presence of debris in engine oil in reasonable time, and the vibration acquisition and analysis part of the system provides an adequate response to artificially induced fault conditions. Findings suggest that

parameters such as spectrogram and RMS are the main features of the signal that can indicate abnormal conditions in stationary industrial equipment.

Although the potential market value of the proposed system might be not sufficient to promote further development of the prototype into a marketable product, the premise and outcomes of the experiments are viable and may be considered as the background to further improvements of the system.

References

1. Agoston, A., Schneidhofer, C., Dörr, N. and Jakoby, B., 2008. A concept of an infrared sensor system for oil condition monitoring.*e & i Elektrotechnik und Informationstechnik*, 125(3), pp.71-5.

2. Broatch, A., Tormos, B., Ruiz, S. and Olmeda, P., 2008. A contribution to the diagnosis of internal combustion engines through rolling block oscillations. *Insight: Non-Destructive Testing & Condition Monitoring*, 50(11), pp.637-41.

3. Brüel and Kjaer, 2009.*Kurtosis in Random Vibration Control*. [online] Brüel and Kjaer Group. Available at:<http://www.bksv.com/controllers> [Accessed 15 February 2017]

4. Cubillo, A., Perinpanayagam, S. and Esperon-Miguez, M., 2016. A review of physics-based models in prognostics: Application to gears and bearings of rotating machinery.*Advances in Mechanical Engineering*, 8(8).

5. Du, L., Zhe, J., Carletta, J., Veillette, R. and Choy, F., 2010. Real-time monitoring of wear debris in lubrication oil using a microfluidic inductive Coulter counting device.*Microfluidics and Nanofluidics*,9(6), pp.1241-5.

6. Dupuis, R., 2010. Application of oil debris monitoring for wind turbine gearbox prognostics and health management. In:*Annual Conference of the Prognostics and Health Management Society, PHM 2010.*[online] Available at:<https://www.scopus.com/inward/record.uri?eid=2-s2.0 84920545843&partnerID=40&md5=beaa001ea054ec1700c92be12e3ea031>[Accessed 20 March 2017]

7. Geng, Z., Chen, J. and Barry Hull, J., 2003. Analysis of engine vibration and design of an applicable diagnosing approach. *International Journal of Mechanical Sciences*, 45(8), pp.1391-410.

8. Howard, I., 1995. VibrationSignal Processing using MATLAB. *Acoustics Australia*, 23, pp.113.

9. Hunt, T., 1995. Monitoring particles in liquids.*Filtration & Separation*, 32(3), pp.205-11.

10. International Organization for Standardization. (2015). *Preferred reference values dor acoustical and vibratory level* (ISO 1683:2015). Available < https://www.iso.org/standard/64648.html> [Accessed 15 February 2017]

11. Kumar, S., Mukherjee, P.S. and Mishra, N.M., 2005. Online condition monitoring of engine oil.*Industrial lubrication and tribology*, 57(6), pp.260-7.

12. Levis, J.W., 2011.*Digital Signal Processing Using MATLAB for Students and Researchers, vol. 1, 1 vols.*.New Jersey: John Wiley & Sons, Inc.

13. Lewicki, D., Blanchette, D. and Biron, G., 1992.*Evaluation of an Oil-Debris Monitoring Device for Use in Helicopter Transmissions*. [online] Available at: <https://www.google.com.au/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=0ahUKEwjWx_q1qY7OAhUFHJQKHcwwDagQFgg_MAU&url=http%3A%2F%2Fntrs.nasa.gov%2Farchive%2Fnasa%2Fcasi.ntrs.nasa.gov%2F19930013637.pdf&usg=AFQjCNFzodI3LZNDtcVUgFwLSFJIUe1xig> [Accessed 25 July 2016].

14. Li, G., Chen, J., Xie, H. and Wang, S., 2016. Vibration test and analysis of mini-tiller. *International Journal of Agricultural & Biological Engineering*, 9(3), pp.97-103.

15. Li, Z., Yan, X., Guo, Z., Liu, P., Yuan, C. and Peng, Z., 2012. A New Intelligent Fusion Method of Multi-Dimensional Sensors and Its Application to Tribo-System Fault Diagnosis of Marine Diesel Engines.*Tribology Letters*, 47(1), pp.1-15.

16. Macián, V., Tormos, B., Olmeda, P. and Montoro, L., 2003. Analytical approach to wear rate determination for internal combustion engine condition monitoring based on oil analysis.*Tribology International*, 36(10), pp.771-6.

17. Matsumoto, K., Tokunaga, T. and Kawabata, M., 2016. Engine Seizure Monitoring System Using Wear Debris Analysis and Particle Measurement.*SAE 2016 World Congress and Exhibition*, (2016-April, April).

18. McGeehan, J.A. and Ryason, P.R., 1999. Million mile bearings: Lessons from diesel engine bearing failure analysis. In:*International Fall Fuels and Lubricants Meeting and Exposition.*Toronto, Canada, 25-28 October 1999.[online] Available at: <http://dx.doi.org/10.4271/1999-01-3576> [Accessed 15 February 2017]

19. Miller, J.L. and Kitaljevich, D., 2000. In-line oil debris monitor for aircraft engine condition assessment. In:*IEEE Aerospace Conference Proceedings*, pp. 49-56,<https://www.scopus.com/inward/record.uri?eid=2-s2.0-0034431589&partnerID=40&md5=6f0ebd201fb404dd4e30ab263c78435c> [Accessed 5 April 2017]

20. Misiti, M., Misiti, Y., Oppenheim, G. and Poggi, J.M., 1996.*Wavelet Toolbox for Use with MATLAB*, The MathWorks, Inc.

21. Moosavian, A., Najafi, G., Ghobadian, B., Mirsalim, M., Jafari, S.M. and Sharghi, P., 2016. Piston scuffing fault and its identification in an IC engine by vibration analysis. *Applied Acoustics*, 102, pp.40-8.

22. Muñoz, M., Moreno, F., Bernal, N., Arroyo, J. and Paniagua, L., 2012. Engine diagnosis method based on vibration and acoustic emission energy. *Insight: Non-Destructive Testing & Condition Monitoring*, 54(3), pp.149-54.

23. O'Reilly, R., Khenkin, A. and Harney, K., 2009. Sonic Nirvana: Using MEMS Accelerometers as Acoustic Pickups in Musical Instruments, *Analog Dialogue*, 43.

24. Peng, Z., Kessissoglou, N.J. and Cox, M., 2005. A study of the effect of contaminant particles in lubricants using wear debris and vibration condition monitoring techniques.*Wear*, 258(11-12), pp.1651-62.

25. Raadnui, S. and Kleesuwan, S., 2005. Low-cost condition monitoring sensor for used oil analysis.*Wear*, 259 (7–12), pp.1502-6.

26. Serridge, M. and Licht, T.R., 1987.*Piezoekectric Accelerometers andVibration Preamplifiers*. Denmark: K Larsen& Son A/S.

27. Stecki, J., 1980. Failure Prediction Using Ferrographic Oil Analysis Techniques. In:*National Conference on Lubrication, Friction and Wear in Engineering(1980 : Melbourne, Vic.): Proceedings of the National Conference on Lubrication, Friction and Wear in Engineering*. Melbourne, Australia, 1980. Barton: Institution of Engineers, Australia, Barton, ACT.

28. Sujatha, C., 2010.*Vibration and acoustics: measurement and signal analysis*. New Delhi: Tata McGraw Hill Education Private Ltd.

29. Taghizadeh-Alisaraei, A., Ghobadian, B., Tavakoli-Hashjin, T., Mohtasebi, S.S., Rezaei-asl, A. and Azadbakht, M., 2016. Characterization of engine's combustion-vibration using diesel and biodiesel fuel blends by time-frequency methods: A case study. *Renewable Energy*, 95, pp.422-32.

30. The Australian Centre for Healthcare Governance, 2016. [online]Available at: <www.healthcaregovernance.org.au> [Accessed 10 October 2016].

31. Tienhaara, H., 2004. Guidelines to Engine Dynamics and Vibration. *Marine News*, 2, pp.20-5.

32. Vališ, D., Žák, L.& Pokora, O., 2015. Failure prediction of diesel engine based on occurrence of selected wear particles in oil.*Engineering Failure Analysis*, 56, pp.501-11.

33. Watzenig, D.S.M. and Steiner, G., 2013. Model-Based Condition and State Monitoring of Large Marine Diesel Engines.

34. Welch, G & Bishop, G 2001, *An Introduction to the Kalman Filter*, University of North Carolina at Chapel Hill Department of Computer Science, Chapel Hill, NC.

35. Yibo, C., Xiaopeng, X., Yan, L. and Tianhuai, D., 2010. Fault Diagnosis of Gear Using Oil Monitoring Samples and Vibration Data. In: J Luo, et al., eds.,*Advanced Tribology: Proceedings of CIST2008 & ITS-IFToMM2008*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 934-9.

36. Zhu, J., Yoon, J.M., He, D., Qu, Y. and Bechhoefer, E, 2013. Lubrication oil condition monitoring and remaining useful life prediction with particle filtering.*International Journal of Prognostics and Health Management*, 4 (SPECIAL ISSUE 2).

37. Zhu, X., Du, L. and Zhe, J., 2015. An integrated lubricant oil conditioning sensor using signal multiplexing.*Journal of Micromechanics and Microengineering*, 25(1).

Appendix A - Project Specification

University of Southern Queensland
Faculty of Mechanical Engineering and Surveying

ENG4111/4112 Research Project
PROJECT SPECIFICATION

FOR : **Mikhail Dashchinskiy**

TOPIC: Early Detection and Prevention of Catastrophic Failures in Industrial Engines

MAJOR: Mechanical Engineering

SUPERVISOR: Dr Ray Malpress

PROJECT AIM: Determine a moment at which a crankshaft bearing begins to rapidly deteriorate due to abnormal conditions and stop the equipment before the catastrophic failure occurs.

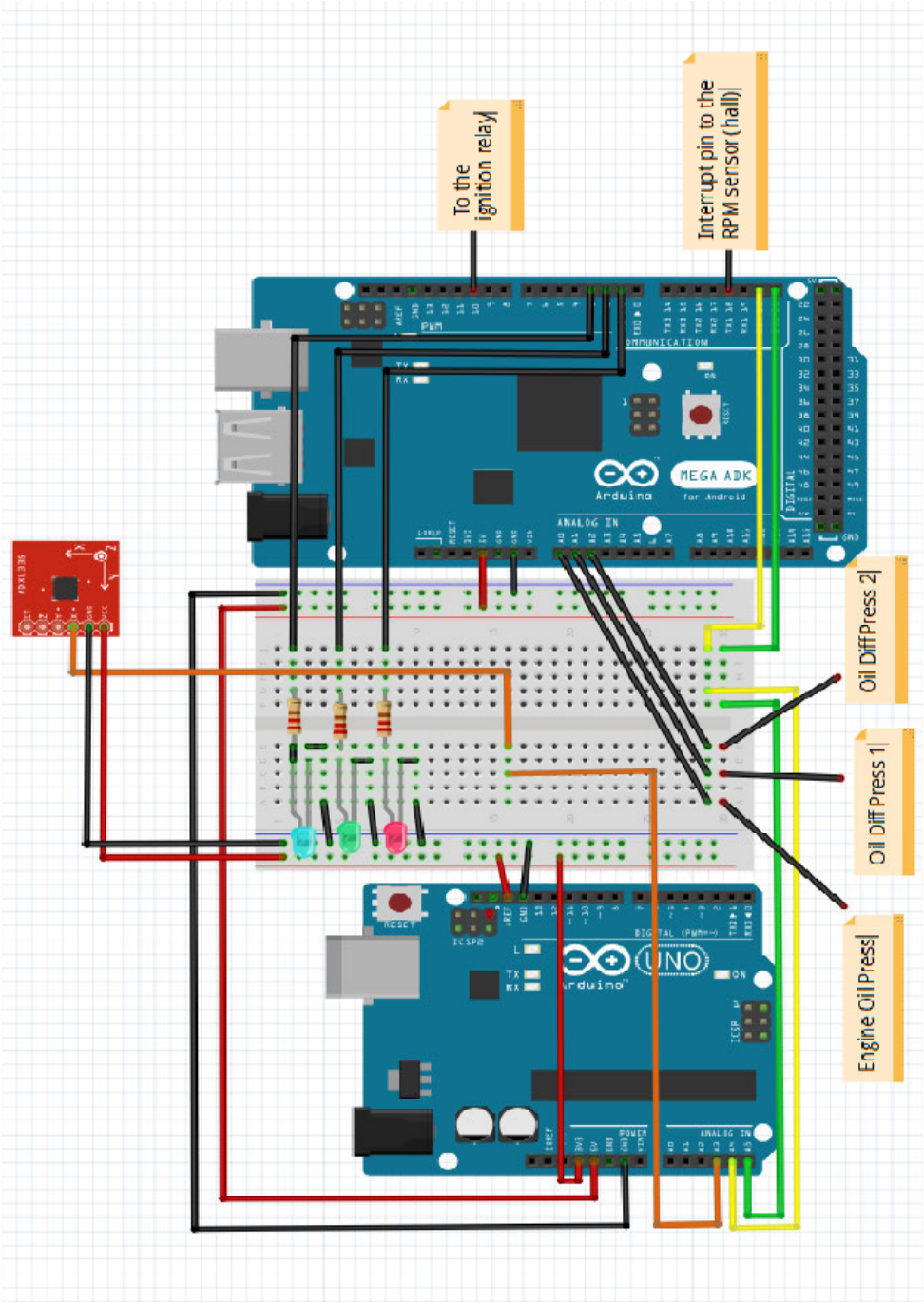
PROGRAMME: Issue D, 6th March 2017

1. Review methods of bearings' failure detection and prevention. Review systems of real time engine condition monitoring being tested or developed.
2. Conceptualize the idea of the engine condition monitoring system using schematics or diagrams.
3. Develop the oil filtering, pressure sensing prototype and the test bench to accommodate the engine and engine supporting systems such as fuel, cooling etc.
4. Review the signal processing and vibration analysis literature and develop software based on MATLAB signal processing toolbox to analyse the vibration data and represent it numerically and graphically.
5. Develop the signal acquisition device based on the Arduino board and the ADXL335 accelerometer.
6. Conduct test runs of two identical engines with all systems incorporated and fault condition induced.
7. Investigate if the system provides sufficient warning of the fault condition by examining resulting damage of the crankshaft. Damage of induced bearing failures will be compared in the engine which was not stopped by the prototype diagnostic system with the engine which was stopped by the prototype system.

Appendix B - Kalman Filter

```
clc
clear all
close all
%% Generate noisy signal
vn=0.2;                                  % The noise amplitude
v=1;                                     % The voltage in volts
f=1000;                                  % The frequency in Hertz
w=2*pi*f;                                % Signal's period
t=linspace(0,5,1000)*1e-3;               % time = 0 to 5ms
sig=v*sin(w*t)+vn*rand(size(t));
                        %% Kalman filter%%
P=1;                                     % Filter sensitivity
R = std(sig);                            % Standard deviation of the signal
K=P/(P+R);                               % Gain coefficient
X = zeros(1,length(sig));                % Filtered signal
P = P*ones(1,length(sig));
for i=2:length(sig)
X(i)=X(i-1);
P(i)=P(i-1);
X(i)=X(i)+K*(sig(i)-X(i));
P(i)=(1-K)*P(i);
end
plot(sig,'r');
hold on
plot(X,'b');
hold off
```

Appendix C - Arduino Boards Connections and Schematics



Appendix D - Matlab Functions and Script Structure.

List of files and toolboxes used by the Matlab GUI.

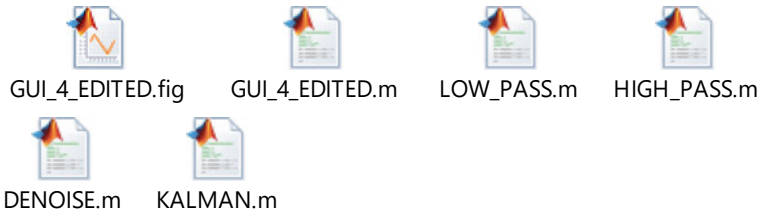
MATLAB Version: 9.0.0.341360 (R2016a)

The following toolboxes must be installed:

Use "ver" command to verify that required toolboxes are available in your system

Control System Toolbox	Version 10.0	(R2016a)
DSP System Toolbox	Version 9.2	(R2016a)
Data Acquisition Toolbox	Version 3.9	(R2016a)
Signal Processing Toolbox	Version 7.2	(R2016a)
Wavelet Toolbox	Version 4.16	(R2016a)

List of files



GUI_4_EDITED.m

```
function varargout = GUI_4_EDITED(varargin)
% GUI_4_EDITED MATLAB code for GUI_4_EDITED.fig

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @GUI_4_EDITED_OpeningFcn, ...
                   'gui_OutputFcn',  @GUI_4_EDITED_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before GUI_4_EDITED is made visible.
function GUI_4_EDITED_OpeningFcn(hObject, eventdata, handles, varargin)

% Choose default command line output for GUI_4_EDITED
handles.output = hObject;
set(handles.SIGNAL_FFT,'String','0');
set(handles.READING_STATUS,'String','OPEN COM FIRST');
set(handles.SEND_WARNING,'String','OPEN COM FIRST');
set(handles.COM_STATUS,'String','OFF LINE');
set(handles.SIGNAL_RMS,'String','0');
set(handles.oil_diff,'String','0');
set(handles.oil_press,'String','0');
set(handles.crest,'String','0');
set(handles.RPM,'String','0');
set(handles.ZERO_ADXL,'Enable','off');
set(handles.SAVE_DATA,'Enable','off');

% % Update handles structure
guidata(hObject, handles);

function varargout = GUI_4_EDITED_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% GROUP ALL DISPLAY WINDOWS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Engine oil pressure display window
function oil_press_Callback(hObject, eventdata, handles)

function oil_press_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end


%% Oil differential pressure display window
function oil_diff_Callback(hObject, eventdata, handles)

function oil_diff_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


%% Crest factor display window
function crest_Callback(hObject, eventdata, handles)

function crest_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% WINDOW TO DISPLAY FFT HZ
function SIGNAL_FFT_Callback(hObject, eventdata, handles)
function SIGNAL_FFT_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% WINDOWS TO DISPLAY SERIAL READ STATUS
function READING_STATUS_Callback(hObject, eventdata, handles)
function READING_STATUS_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% WINDOWS TO DISPLAY PUMP STATUS
function SEND_WARNING_Callback(hObject, eventdata, handles)
function SEND_WARNING_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% WINDOWS TO DISPLAY COM PORT STATUS
function COM_STATUS_Callback(hObject, eventdata, handles)
function COM_STATUS_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% WINDOW TO DISPLAY SIGNAL RMS
function SIGNAL_RMS_Callback(hObject, eventdata, handles)
function SIGNAL_RMS_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),...
```

```
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% WINDOW TO DISPLAY RPM
function RPM_Callback(hObject, eventdata, handles)
function RPM_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% GROUP ALL RADIOBUTTONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%% ENABLE COM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function OPEN_COM_Callback(hObject, eventdata, handles)
button_state = get(hObject,'Value');
log_state=get(handles.PROCESS_DATA_LOG,'Value');
if log_state==0
set(handles.OPEN_COM,'Enable','on');
if button_state == 1 && log_state ==0
set(handles.COM_STATUS,'String','INITIALIZE');
delete(instrfindall);
try
port=char(handles.port);
s1 = serial(port);
catch
    err_str = {'YOU HAVE NOT CONNECTED ANY DEVICE',...
        'OR HAVE NOT SELECTED THE COM PORT'};
    ed = errordlg(err_str,'Error');
set(ed, 'WindowStyle', 'modal');
uiwait(ed);
port=0;
s1=0;
set(handles.OPEN_COM,'Value',0);
return;
end
s1.StopBits=1;
s1.DataBits=8;
s1.BaudRate=115200;
% set(s1,'Terminator', 'Z');
try
buffer=eval(char(handles.choise));  % The size of the buffer is set
% according to the user's selection
catch
    buffer=2000;
    ed = errordlg('The buffer set to default size 2000','Warning');
    set(ed, 'WindowStyle', 'modal');
end
set(s1, 'InputBufferSize', buffer);
handles.s1=s1;
guidata(hObject,handles);
pause(1);                    % Give the Arduino time to reboot
fopen(s1);
s1.ReadAsyncMode = 'continuous';
readasync(s1);
pause(2);
% Radio button status
set(handles.COM_STATUS,'String','ON LINE');
```

```
set(handles.STOP_ENG,'Enable','on');
set(handles.ZERO_ADXL,'Enable','on');
% Edit text status
set(handles.READING_STATUS,'String','Zero ADXL First');
set(handles.SEND_WARNING,'String','OFF');
else
    set(handles.OPEN_COM,'Value',0);
    set(handles.COM_STATUS,'String','OFF LINE');

end
if button_state == 0
    set(handles.READ_SERIAL,'Value',0);
    set(handles.STOP_ENG,'Value',0);
    set(handles.SAVE_DATA,'Value',0);
    set(handles.COM_STATUS,'String','OFF LINE');
    set(handles.READ_SERIAL,'Enable','off');
    set(handles.STOP_ENG,'Enable','off');
    set(handles.ZERO_ADXL,'Enable','off');
    set(handles.READING_STATUS,'String','OPEN COM FIRST');
    set(handles.SEND_WARNING,'String','OPEN COM FIRST');
    set(handles.SIGNAL_FFT,'String',0);
    set(handles.SIGNAL_RMS,'String',0);
    set(handles.SAVE_DATA,'Enable','off');
    pause(1);
    cla(handles.axes1);
    cla(handles.axes2);
    cla(handles.axes3);
    delete(instrfindall);
end
else
    set(handles.OPEN_COM,'Enable','off');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ONLINE DATA PROCESSING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Enable FFT spectrum plot
function enable_FFT_Callback(hObject, eventdata, handles)

% Kalman Filter
function kalman_Callback(hObject, eventdata, handles)

% ENABLE PLOTTING CREST FACTOR AND RMS
function crest_RMS_Callback(hObject, eventdata, handles)
handles.axes3;
cla;


%%%%%%%%%%%%%%%%%%%%%%%% READ SERIAL PORT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function READ_SERIAL_Callback(hObject, eventdata, handles)
button_state = get(hObject,'Value');
% handles.Read_Serial=button_state;
% guidata(hObject,handles);
clc
set(handles.READING_STATUS,'String','READING');
if button_state == 1
    set(handles.SAVE_DATA,'Enable','on');
while (button_state > 0);
```

```
    % Get states of radio buttons
button_state = get(hObject,'Value');
button_state_3=get(handles.OPEN_COM,'Value');
button_state_9=get(handles.SAVE_DATA,'Value');
button_state_10=get(handles.enable_FFT,'Value');
if button_state == 0 && button_state_3 == 1
    set(handles.READING_STATUS,'String','OFF');
    break
elseif button_state == 0 && button_state_3 == 0
    set(handles.READING_STATUS,'String','OPEN COM FIRST');
    break
end
%% VERSION WITH "fread" IMPROVED PERFORMANCE
a_in =fread(handles.s1)';   % Read sensor
a_in=char(a_in);            % Convert input string to characters array

% GET SENSORS DATA OUT OF THE DATA STRING
% In this section the main string contains vibration data with included
% RPM readings
% value. So we are catching this RPM value then exclude it from the
% data string.
% Then we combine the parts of the string in one string and send it for
% further processing. The sensors readings are separated by
% 'Z'characters in the data string.

     expression = 'Z';            % Define the delimeter character
     % Split by the patern
     [match,noMatch]=regexp(a_in,expression,'match','split');
     assignin('base','val',noMatch);
     try
     RPM=noMatch(2);                        % Catch up the RPM value
     RPM=eval(cell2mat(RPM));

     OIL=noMatch(3);                        % Catch up the OIL press
     OIL=eval(cell2mat(OIL));

     DIFF=noMatch(4);                       % Catch up the OIL press
     DIFF=eval(cell2mat(DIFF))-9;

     % Combine data to be saved in csv file
     Sens_data=[RPM,OIL,DIFF];

     % Display sensors data
     set(handles.RPM,'String',num2str(RPM)); % Display RPM
     set(handles.oil_press,'String',num2str(OIL)); %Display Oil press
     set(handles.oil_diff,'String',num2str(DIFF)); %Display Oil press
     catch
     % If DATA data is empty
     RPM=0;
     OIL=0;
     DIFF=0;
     Sens_data=0;
     end
   try
   % Concatinate the vibration data string
     splitStr = regexp(a_in,'\,','split');%Split the string onto values
   catch
     splitStr = regexp(a_in,'\,','split');   % If RPM data is empty
   end
     splitStr = splitStr(10:end-10);       % Remove spikes in the data
 % Convert characters into numerical array for further processing
 z=0;
    for k=1:(length(splitStr))
```

```
        b=cell2mat(splitStr(k));
        z=z+1;
try
         a(z)=eval(b)-handles.calibval;    % Substruct calibration value
catch
         z=z-1; % In case of data error (jitter)
end
    end
  a=a(1:end-1);
%% WRITE DATA TO THE FILE
if button_state_9 == 1
dlmwrite('FRIQUENCY.csv',a,'-append');
dlmwrite('FRIQUENCY.csv',Sens_data,'-append');
end

%% FFT TRANSFORMATION
if button_state_10 == 1
fs=2900; % Sampling frequiency
nf=2900; % number of point in DTFT
%%
Y = fft(a,nf);%,nf);
f = fs/2*linspace(0,1,nf/2+1);
Y=abs(Y);
% Plot FFT
[pk,MaxFreq]=findpeaks(abs(Y(1:nf/2+1)));
b=max(pk);
c=MaxFreq(find(pk==max(pk)));
set(handles.SIGNAL_FFT,'String',['Max ',num2str(c),' Hz']);
plot(handles.axes1,f,abs(Y(1:nf/2+1)),c,b,'o',[RPM/60 RPM/60],[1 b],'r-
');
xlim(handles.axes1,[0 400]);
title(handles.axes1,'FTT');
xlabel(handles.axes1,'Frequency');
ylabel(handles.axes1,'Magnitude');
set(handles.RPM,'String',['RPM= ', num2str(RPM)]); % Display RPM
drawnow
end
% Plot raw signal
plot(handles.axes2,a);
title(handles.axes2,'PURE SIGNAL');
xlabel(handles.axes2,'N samples');
ylabel(handles.axes2,'Magnitude');
drawnow
end
set(handles.WATERFALL,'Enable','On');
else set(handles.SAVE_DATA,'Enable','off');
end


% ENGINE STOP COMMAND TO BE SENT TO ARDUINO
function STOP_ENG_Callback(hObject, eventdata, handles)
button_state=get(hObject,'Value');
if button_state == 1
        set(handles.SEND_WARNING,'String','WARNING');
        fprintf(handles.s1,'%i',1,'async');
        flushinput(handles.s1);
        pause(1);
elseif button_state==0
        set(handles.SEND_WARNING,'String','No WARNING');
        fprintf(handles.s1,'%i',0,'async');
        pause(1);
        flushinput(handles.s1);
```

```
end

% LOW PASS FILTER
function LOW_PASS_Callback(hObject, eventdata, handles)

% HIGH PASS FILTER
function HIGH_PASS_Callback(hObject, eventdata, handles)

% WAVELET DENOISE 1
function Wavelet_1_Callback(hObject, eventdata, handles)

% WAVELET DENOISE 2
function Wavelet_2_Callback(hObject, eventdata, handles)

% PLOT SPECTROGRAM
function PSD_SPECTROGRAM_Callback(hObject, eventdata, handles)
% Since RMS/CREST plot and Spectrogram plot shares the same axes. The
% RMS/CREST plot button must be disabled.
if get(hObject,'Value') == 0
  set(handles.crest_RMS,'Enable','on');
else
  set(handles.crest_RMS,'Enable','off');
  set(handles.crest_RMS,'Value',0);
end


% SAVE DATA
function SAVE_DATA_Callback(hObject, eventdata, handles)
button_state=get(hObject,'Value');
pause(1);
% When user decides to finish writing the data GUI will ask to save the
% file.
if button_state == 0
[file,path] = uiputfile('*.csv','Save as'); % Initiates "save as"
% dialog and  returns user selected filename and the path
try
log=csvread('FRIQUENCY.csv'); % Reads current vibration data file
csvwrite([path,file],log);    % Saves to specified folder under
specified
% filename
catch  % If user hits cancel
err_str = {'NO DATA HAS BEEN SAVED. THE DATA STILL EXISTS UNDER',...
    '"FRIQUENCY.csv" BUT WILL BE LOST AFTER NEXT RECORDING'};
ed = errordlg(err_str,'Error');
set(ed, 'WindowStyle', 'modal');
return;
end
end
if exist('FRIQUENCY.csv','file')==2
   delete('FRIQUENCY.csv');  % Delete peviously recorded data
end

% GET RMS OF THE SIGNAL
function INCLUDE_RMS_Callback(hObject, eventdata, handles)

% CALIBRATE ADXL
% This function used to zero reference the ADXL accelerometer
function ZERO_ADXL_Callback(hObject, eventdata, handles)
button_state=get(hObject,'Value');
if button_state==1;
    set(handles.READ_SERIAL,'Enable','off');
    set(handles.READ_SERIAL,'Value',0);
    set(handles.READING_STATUS,'String','Zero ADXL');
```

```
    fprintf(handles.s1,'%i',2);          % Warm up the hardware
    pause(1);
    fprintf(handles.s1,'%i',3);        % Initialize ADXL calibration
    pause(1);
    flushinput(handles.s1);
  a_in =fread(handles.s1)';
     a_in=char(a_in);
     splitStr = regexp(a_in,'\,','split');
     splitStr=splitStr(250);            % Pick mid value
       b=cell2mat(splitStr);
try
           calibval=eval(b);
catch
          calibval=0; % In case of data error (jitter)
end
    fprintf(handles.s1,'%i',2);          % Enable sensor reading
    handles.calibval=calibval;
    guidata(hObject,handles);
    pause(1);
    %%
    set(handles.READ_SERIAL,'Enable','on');
    if handles.READ_SERIAL == 1
    set(handles.READING_STATUS,'String','READING');
    else
    set(handles.READING_STATUS,'String','NOT READING');
    end
    set(hObject,'Value',0);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOGGED DATA OFFLINE PROCESSING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function PROCESS_DATA_LOG_Callback(hObject, eventdata, handles)
button_state=get(hObject,'Value');
if button_state==1;
%     tic;
    set(handles.OPEN_COM,'Value',0);
    set(handles.READ_SERIAL,'Value',0);
    set(handles.STOP_ENG,'Value',0);
    set(handles.SAVE_DATA,'Value',0);
    set(handles.READ_SERIAL,'Enable','off');
    set(handles.STOP_ENG,'Enable','off');
    set(handles.OPEN_COM,'Enable','off');
    set(handles.SAVE_DATA,'Enable','off');
    set(handles.ZERO_ADXL,'Enable','off');
    set(handles.WATERFALL,'Enable','off');
    set(handles.READING_STATUS,'String','OPEN COM FIRST');
    set(handles.SEND_WARNING,'String','OPEN COM FIRST');
    set(handles.SIGNAL_FFT,'String',0);
    set(handles.SIGNAL_RMS,'String',0);
    pause(1);
    cla(handles.axes1);
    cla(handles.axes2);
    cla(handles.axes3);

% GET THE DATA BLOCK
% Check if logged data exist and read the file
    try
        file=handles.file;
        log=csvread(file);
```

```
    catch
        set(hObject,'Value',0);
        set(handles.OPEN_COM,'Enable','on');
        set(handles.COM_STATUS,'String','OFF LINE');
        ed = errordlg('Open file first','Error');
        set(ed, 'WindowStyle', 'modal');
        log=0;
        return;
    end
% Process data
[r,c]=size(log);
%Remove empty data before and after the engine start
% All data before 1500 RPM achieved will be erased
start=find(log(1:r,:)>=1500);
log(1:start(1),:)=[];
log(start(end):end,:)=[];

% Separate Sensors data from vibration data.
[r,c]=size(log);
Sens_data=log;
Sens_data(1:2:r,:)=[];                          % Remove vibration data
% Sens_data=Sens_data(find(Sens_data));          % Combine sensors data
log(2:2:r,:)=[];                              % Remove sensors data
for i=1:r/2
button_state=get(hObject,'Value');
 % Get states of radio buttons
button_state_4=get(handles.LOW_PASS,'Value');
button_state_5=get(handles.HIGH_PASS,'Value');
button_state_6=get(handles.Wavelet_1,'Value');
button_state_7=get(handles.Wavelet_2,'Value');
button_state_8=get(handles.PSD_SPECTROGRAM,'Value');
button_state_9=get(handles.kalman,'Value');
button_state_10=get(handles.INCLUDE_RMS,'Value');
button_state_11=get(handles.PAUSE,'Value');
button_state_12=get(handles.crest_RMS,'Value');
button_state_13=get(handles.enable_FFT,'Value');

                % Remove erratic data
                if i>1
                  if sum(log(i,:))>10000
                    log(i,:)=log(i-1,:);
                  end
                end
                %
        a=log(i,:); % Get vibration data
        RPM(i)=Sens_data(i,1); % Get RPM
        % Ignor errors in RPM reading
        if i>1
            if RPM(i)<RPM(i-1)-1000 || RPM(i)>RPM(i-1)+1000
                RPM(i)=RPM(i-1);
            end
        end

        OIL(i)=Sens_data(i,2); % Get Oil Pressure
        DIFF(i)=Sens_data(i,3); % Get Oil Differential pressure

% Stop offline data processing if button is un-pressed
if button_state == 0
    break;
end

if button_state_9 == 1
P = 1; % Kalman filter sensitivity
```

```
a = KALMAN(a,P);    % Call the Kalman filter
end

%% PAUSE THE EXECUTION
if button_state_11 == 1
while (button_state_11 > 0);
    pause(1);
    button_state_11=get(handles.PAUSE,'Value');
end
end

%% SHOW RMS and CREST factor
    RMS(i)=rms(a);
    CREST(i)=peak2rms(a); % Calculate Crest Factor
    k(i) = kurtosis(a);   % Calculate kurtosis
    % Correct errors in data to avoid spikes
    if i>1
    if RMS(i)>RMS(i-1)+4 || CREST(i)>CREST(i-1)+4
        RMS(i)=RMS(i-1);
        CREST(i)=CREST(i-1);
    end
    end
    set(handles.crest,'String',num2str(CREST(i)));
    set(handles.SIGNAL_RMS,'String', num2str(RMS(i)));

%% PLOT RMS AND CREST FACTOR
if button_state_12 == 1 && button_state_8 == 0
    handles.axes3;
    cla;
    title('Real time RMS,CREST,RPM and PRESS DIFF');
    plot(CREST);
    ylim([0 15]);
    xlim([0,100]);
    if button_state_10 == 1
        handles.axes3;
        hold on;
        plot(RMS);
        plot(RPM/1000);
        plot(DIFF/100);
    end
    if i>100;
    xlim([i-100,i]);
    end

else
    handles.axes3;
    hold off;
end
%% WAVELET DENOISE FILTERS
if button_state_6 == 1
a=DENOISE(a); % DENOISE USING WAVELET.
end
if button_state_7 == 1
[C,L] = wavedec(a,3,'db3'); %% Three levels of denosing
a = wrcoef('a',C,L,'db3',3); %% Reconstruct the signal
end

%% LOW PASS AND HIGH PASS FILTERS
if button_state_4 == 1
a=filter(LOW_PASS,a); % BATERWOTH PRE_FILTER LOW_PASS
end
if button_state_5 == 1
a=filter(HIGH_PASS,a); % CHEBUSHEV PRE_FILTER HIGH_PASS
```

```
end


%% FFT Transformation and Spectrum plot
fs=2900;
nf=2900; %number of point in DTFT
Y = fft(a,nf);%,nf);
f = fs/2*linspace(0,1,nf/2+1);
Y=abs(Y);
% Plot FFT
if button_state_13 == 1;
[pk,MaxFreq]=findpeaks(abs(Y(1:nf/2+1)));
b=max(pk);
c=MaxFreq(find(pk==max(pk)));
set(handles.SIGNAL_FFT,'String',['Max ',num2str(c),' Hz']);
plot(handles.axes1,f,abs(Y(1:nf/2+1)),c,b,'o',[RPM(i)/60 RPM(i)/60],...
    [1 b],'r-',[RPM(i)/30 RPM(i)/30],[1 b],'r-',...
    [RPM(i)/20 RPM(i)/20],[1 b],'r-');
xlim(handles.axes1,[0 400]);
ylim(handles.axes1,[0 3000]);
title(handles.axes1,'FFT');
xlabel(handles.axes1,'Frequency');
ylabel(handles.axes1,'Magnitude');
end

% Display sensors data
set(handles.RPM,'String',num2str(RPM(i))); % Display RPM
set(handles.oil_press,'String',num2str(OIL(i))); %Display Oil press
set(handles.oil_diff,'String',num2str(DIFF(i))); %Display Oil press

% Plot raw signal
plot(handles.axes2,a);
title(handles.axes2,'PURE SIGNAL');
xlabel(handles.axes2,'N samples');
ylabel(handles.axes2,'Magnitude');
ylim(handles.axes2,[-25 25]);

% Plot PSD periodogram
if button_state_8 == 1;
set(handles.crest_RMS,'Enable','off');
set(handles.crest_RMS,'Value',0);
axes(handles.axes3);
cla;
plot(psd(spectrum.periodogram,a,'Fs',2900,'NFFT',length(a)));
xlim(handles.axes3,[0.01 0.4]);
ylim(handles.axes3,[-20 20]);
end
drawnow

end
        set(hObject,'Value',0);
        set(handles.OPEN_COM,'Enable','on');
        set(handles.COM_STATUS,'String','OFF LINE');
else
    set(handles.COM_STATUS,'String','OFF LINE');
    set(handles.OPEN_COM,'Enable','on');
    set(handles.SAVE_DATA,'Enable','on');
    set(handles.WATERFALL,'Enable','on');
end

% PAUSE OF THE LOGGED DATA
function PAUSE_Callback(hObject, eventdata, handles)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% GENERATED PLOTS BASED ON RECORDER DATA %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% PLOT SPECTRUM DENSITY OF LOGGED DATA "WATERFALL"
function WATERFALL_Callback(hObject, eventdata, handles)
button_state=get(hObject,'Value');
kalman = get(handles.kalman,'Value');
wavelet_1 = get(handles.Wavelet_1,'Value');
wavelet_2 = get(handles.Wavelet_2,'Value');
if button_state == 1
    try
        file=handles.file;
        log=csvread(file);
        [r,c]=size(log);

%Remove empty data before and after the engine start
% All data before 1500 RPM achieved will be erased
start=find(log(1:r,:)>=1500);
log(1:start(1),:)=[];
log(start(end):end,:)=[];

        [r,c]=size(log);
        log(2:2:r,:)=[];  % Remove RPM
    catch
        set(hObject,'Value',0);  % In case if the file does not exist
        ed = errordlg('Open file first','Error');
        set(ed, 'WindowStyle', 'modal');
        return;
    end
% Clean up errors from the data
        [r,c]=size(log);
    for i=1:r

       if i>1
            if sum(log(i,:))>10000
                    log(i,:)=log(i-1,:);
            end
       end

            % Apply Kalmnn filter to the data
       if kalman == 1;
          a = log(i,:);
          P = 1;
          log(i,:) = KALMAN(a,P);
       end

            % WAVELET DENOISE
       if wavelet_1 == 1;
          log(i,:)=DENOISE(log(i,:)); % DENOISE USING WAVELET.
       end
      if wavelet_2 == 1;
        [C,L] = wavedec(log(i,:),3,'db3'); %% Three levels of denosing
        log(i,:) = wrcoef('a',C,L,'db3',3); %% Reconstruct the signal
       end

       end
% Make a single row array out of matrix.
% Remember to transpose matrix "log'"
OUT=reshape(log',1,[]);
OUT=OUT';
```

```
Seg=c; % Size of segment of data.
Fs=2900; % SAMPLE FRIQUENCY
OvLap=200; % Overlap between samples
window = hamming(Seg);
[S, F, T, P] = spectrogram(OUT, window, OvLap, Seg,
Fs,'MinThreshold',100);
fig=figure;
mesh(T,F,abs(S));
view([-90.3 90])
ylim([10 300]);
% zlim([0 5000]);
title('SPECTROGRAM');
xlabel('Time (sec)');
ylabel('Friquency (Hz)');
xlim auto;
rotate3d on;
figure(fig);
else
    cla(handles.axes1);
    rotate3d off;
end

%% Plot CREST_RMS_RPM.
function CREST_RMS_RPM_Callback(hObject, eventdata, handles)
%% PLot CREST,RMS, Pressure rise and RPM in separate window.
% This is the plot that can be called to reveal data recorded during
% the experiment. It does not work in real time signal reading. This
% plot can be used for determining a relation between plotted
% parameters.

button_state=get(hObject,'Value');
kalman = get(handles.kalman,'Value');
wavelet_1 = get(handles.Wavelet_1,'Value');
wavelet_2 = get(handles.Wavelet_2,'Value');
if button_state == 1
    try
        file=handles.file;
        log=csvread(file);
        [r,c]=size(log);

%Remove empty data before and after the engine start
% All data before 1500 RPM will be erased
start=find(log(1:r,:)>=1500);
log(1:start(1),:)=[];
log(start(end):end,:)=[];
        [r,c]=size(log);
        Sensors=log;
        Sensors(1:2:r,:)=[];  % Remove vibration data
        log(2:2:r,:)=[];  % Remove RPM
    catch
        set(hObject,'Value',0);  % In case if the file does not exist
        ed = errordlg('Open file first','Error');
        set(ed, 'WindowStyle', 'modal');
        log = 0;
        return;
    end
% Clean up errors from the data
        [r,c]=size(log);
        for i=1:r
            RPM(i)=Sensors(i,1);
            OIL(i)=Sensors(i,3);
% Remove data errors
            if i>1
```

```
                if sum(log(i,:))>10000
                    log(i,:)=log(i-1,:);
                end
                if RPM (i)<RPM(i-1)-1000;
                   RPM(i)=RPM(i-1);
                end
            end


            % Apply Kalmann filter to the data
            if kalman == 1;
            a = log(i,:);
            P = 1;
            log(i,:) = KALMAN(a,P);
            end

            % WAVELET DENOISE
            if wavelet_1 == 1;
            log(i,:)=DENOISE(log(i,:)); % DENOISE USING WAVELET.
            end
            if wavelet_2 == 1;
            [C,L] = wavedec(log(i,:),3,'db3');
            log(i,:) =wrcoef('a',C,L,'db3',3);
            end

RMS(i) = rms(log(i,:));
CREST(i) = peak2rms(log(i,:));
        end

figure(1);
hold on
title('Signal Crest Factor and RMS');
% Kalman filter for the global data
P=0.05;
P1=0.05;
P2=1;
X = KALMAN(CREST,P);
X1 = KALMAN(RMS,P1);
X2 = KALMAN(OIL,P2);
plot(X);
plot(X1);
plot(RPM/1000);
plot(X2/50);
ylim([0 20]);
legend('CREST Factor','RMS','RPM','OIL PRESSURE');
hold off
else
    close(figure(1));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% TOOLBAR MENU %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%% FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function file_menu_Callback(hObject, eventdata, handles)

% GET HELP FILE
function help_Callback(hObject, eventdata, handles)
winopen GUI_HELP.txt;
```

```
% SELECT PORT FOR THE MICROCONTROLLER
% When user presses 'port' item in the toolbar menu the function
% initiates a COM port selection dialog and passes selected port to the
% OPEN COM function.

function select_port_Callback(hObject, eventdata, handles)
list = instrhwinfo('serial');       % Get available COM ports
if isempty(list.SerialPorts) == 0   % Check if anything plugged to USB
for i=1:length(list.SerialPorts)
ports(i) = list.SerialPorts(i); %Create a cell array of available ports
end
% Create list dialog. s-return the number of selected component from
% the ports cell array. v-returns 1 if selection was made and 0 if
% no selection.
[s,v]=listdlg('PromptString','Select COM
port','SelectionMode','single',...
    'ListString',ports);
if v==1
handles.port=char(ports(s));
guidata(hObject, handles);
else
handles.port={};
guidata(hObject, handles);
end
else
set(handles.port,'Label','No device connected');
end

% --------------------------------------------------------------------
function port_Callback(hObject, eventdata, handles)


% OPEN  FILE
% This function initiates a browse window to navigate to
% the vibration data *.csv file

function open_file_Callback(hObject, eventdata, handles)
file = uigetfile('*.csv');
handles.file=file;
guidata(hObject, handles);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% POPUP MENUES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Select Buffer Size POPUP menu
function buffer_Callback(hObject, eventdata, handles)
content=cellstr(get(hObject,'String'));
choise=content(get(hObject,'Value'));
handles.choise=choise;
guidata(hObject, handles);

function buffer_CreateFcn(hObject, eventdata, handles)

%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,...
        'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

LOW_PASS.m

```
function Hd = LOW_PASS
%LOW_PASS Returns a discrete-time filter object.

% MATLAB Code
% Generated by MATLAB(R) 9.0 and the DSP System Toolbox 9.2.
% Generated on: 26-Dec-2016 13:19:40

% Butterworth Lowpass filter designed using FDESIGN.LOWPASS.

% All frequency values are in Hz.
Fs = 2900;  % Sampling Frequency

Fpass = 150;         % Passband Frequency
Fstop = 200;         % Stopband Frequency
Apass = 1;           % Passband Ripple (dB)
Astop = 80;          % Stopband Attenuation (dB)
match = 'passband';  % Band to match exactly

% Construct an FDESIGN object and call its BUTTER method.
h  = fdesign.lowpass(Fpass, Fstop, Apass, Astop, Fs);
Hd = design(h, 'butter', 'MatchExactly', match);

% [EOF]
```

HIGH_PASS.m

```
function Hd = HIGH_PASS
%HIGH_PASS Returns a discrete-time filter object.

% MATLAB Code
% Generated by MATLAB(R) 9.0 and the DSP System Toolbox 9.2.
% Generated on: 26-Dec-2016 13:41:52

% Chebyshev Type II Highpass filter designed using FDESIGN.HIGHPASS.

% All frequency values are in Hz.
Fs = 2900;  % Sampling Frequency

N     = 100;  % Order
Fstop = 5;   % Stopband Frequency
Astop = 80;   % Stopband Attenuation (dB)

% Construct an FDESIGN object and call its CHEBY2 method.
h  = fdesign.highpass('N,Fst,Ast', N, Fstop, Astop, Fs);
Hd = design(h, 'cheby2');
```

DENOISE.m

```
function sigDEN = DENOISE(SIG)
% FUNC_DENOISE_DW1D Saved Denoising Process.
%   SIG: vector of data
%   -------------------
%   sigDEN: vector of denoised data

%  Auto-generated by Wavelet Toolbox on 11-Jan-2017 19:40:07

% Analysis parameters.
%---------------------
wname = 'db10';
level = 3;

% Denoising parameters.
%----------------------
% meth = 'sqtwolog';
% scal_or_alfa = one;
sorh = 's';    % Specified soft or hard thresholding
thrSettings =  [...
    5.231591207698597 ; ...
    10.052555720277580 ; ...
    4.543068930466854   ...
    ];

% Denoise using CMDDENOISE.
%--------------------------
sigDEN = cmddenoise(SIG,wname,level,sorh,NaN,thrSettings);
```

KALMAN.m

```
%% The Kalman filter.
% This fucntios is fritten based on the literature review.
% Matlab has its own  "kalman" function which I consider to be
% a little bit complicated for a simple denoise.
% The aray a contains the data, P determines the filter's
% sensitivity. X is is the denoised data.
function X = KALMAN(a,P)
X = zeros(1,length(a));       % Filtered signal
R = std(a);                   % Standard deviation of the signal
K=P/(P+R);                    % Gain coefficient
P = P*ones(1,length(a));
for j=2:length(a)
    X(j)=X(j-1);
    P(j)=P(j-1);
    X(j)=X(j)+K*(a(j)-X(j));
    P(j)=(1-K)*P(j);
end
```

Appendix E - C Program Structure

The Master Board Script

```
////////////////////////////////////////////////////////////////////////////
//////////////////////THE MASTER BOARD SCRIPT/////////
////////////////////////////////////////////////////////////////////////////

// The following script has to be loaded into the master board
// it controls communication between the master, the slave board and PC
// by means of I2C protocol and serial COM interface. Also the master board is
// responsible for queering the ADSL accelerometer, gathering all data received
// from the slave board in send it to PC.

#include <Wire.h> // Header file used for I2C communication

// DECLARATIONS
int zPin = A3; // Z axis of the ADXL335
int block=500; // Block of data
unsigned int  rawZ;  // DATA FROM ADXL
int ave;

// void EVENT declarations
boolean DATA_ON = false; // SENT DATA
boolean CALIB_DONE = false; // CALIBRATION VERIFICATION

// OVERCLOCK ADC. Set bits that control prescaler of the ADC clock
// Refer to ATmega328 Specifications.
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))

////////////////////////////////////////////////////////////////////
//////////////////////////  INITIAL SETUP //////////////////
////////////////////////////////////////////////////////////////////

void setup(){

  analogReference(EXTERNAL);  // Reference the ADC to external 3.5 V
  Serial.begin(115200);    // Begin serial communication at 115200 baud rate
  Wire.begin();        // Join I2C bus (address optional for master)

 //OVERCLOCK ADC .Set the prescaler value 16 which gives Fs=70kHz
// Refer to ATmega328 Specifications.

 sbi(ADCSRA, ADPS2);
 cbi(ADCSRA, ADPS1);
 cbi(ADCSRA, ADPS0);
}


////////////////////////////////////////////////////////////////////////////
////////////////////////////END OF INITIAL SETUP///////////
////////////////////////////////////////////////////////////////////////////
```

```
// This is a subroutine initiated by Matlab scrip to zero reference the ADXL accelerometer

void calibrate(){
// Declare Variables
ave = 0;
unsigned int zCalib = 0;

// If Boolean CALIB_DONE is false the board will read number of samples
// specified by the variable' block' and then average the block of data.

if (!CALIB_DONE) {
for (int i=0; i<block; i++){
 zCalib=analogRead(zPin);  // Read the voltage coming from the Z axis of
//  the ADXL
 ave+=zCalib;            // Make cumulative sum of the readings.

//DEBUG (Uncomment for debugging)
 //Serial.println(zCalib);
 //Serial.println(ave);
 }
ave=ave/block;  // Average the block of data
}

 CALIB_DONE = true; // Finish reading the accelerometer and toggle the
// Boolean
// The function will continue to send calibration data to PC until it receives
// the response setting the Boolean DATA_ON true.

 while (DATA_ON == false){
 Serial.print(ave);
 Serial.print(',');
 serialEvent();
 }
 delay(1000);  // Provide some delay to accommodate for latencies
}

////////////////////////////////////////////////////////////////////////////
////////////////// THE MAIN LOOP////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void loop(){

 if (DATA_ON & CALIB_DONE){
  // PRINT BLOCK OF DATA
// The function will send a number of consecutive reading specified
// by the 'block' variable

  for (int i=0; i<block; i++){
    rawZ=analogRead(zPin);   // Read the ADXL
    Serial.print(rawZ);            // Send the vibration data to PC
    Serial.print(',');              // Send delimiter character
  }
```

```
    getRPM();  // Launch the subroutine that takes care of receiving the
// sensors data from the slave board and sending them to PC
 }
}

////////////////////////////////////////////////////////////////////////////
////////// END OF THE MAIN LOOP//////////////////////////
////////////////////////////////////////////////////////////////////////////

// REACT ON COMMANDS  COMING FROM MATLAB
// This function interrupts the main routine when data from PC is
// available at the serial port. The function will generate four different
// responses according to received integer from PC

void serialEvent(){
  byte m = 1;
  byte n = 0;

  switch (Serial.read()) {  // Read a character from serial port

    case '1':  //  Send Warning to slave
      // This part is used to sent data to slave to get reaction
       Wire.beginTransmission(8);
       Wire.write(m);       // Send character 'm' to the slave
       Wire.endTransmission();
      break;

    case '0':  //  Call off Warning
       Wire.beginTransmission(8);
       // This part is used to sent data to slave to get reaction
       Wire.write(n);      // Send character 'n' to the slave
       Wire.endTransmission();
      break;

    case '2': // READY TO RECEIVE DATA
      DATA_ON = true;
      break;

    case '3': // CALIBRATE ADXL
      CALIB_DONE = false;
      calibrate();
    break;
  }
  while (Serial.read() >= 0); // flush any extra characters
}

// This subroutine reads data from the slave board, combine one byte words
// into two byte words. Then it sends the data to PC through the serial port.

void getRPM() {

  // Request RPM data from the slave
  Wire.requestFrom(8, 6);    // request 6 bytes from slave device #8
```

```
 while (Wire.available()) {
   //I2C can only send one byte per cycle. So for long numbers
   // we need to split them in two bytes
   int RPM,OIL,Diff; // Declare the variables
   byte a, b,c,d,e,f;  // Declare the bytes characters
   a = Wire.read();
   b = Wire.read();
   c = Wire.read();
   d = Wire.read();
   e = Wire.read();
   f = Wire.read();

   RPM = a; // Get first byte
   RPM = (RPM << 8) | b; // Bitwise shift left and add second byte

   OIL = c; // Get first byte
   OIL = (OIL << 8) | d; // Bitwise shift left and add second byte

   Diff = e; // Get first byte
  Diff = (Diff << 8) | f; // Bitwise shift left and add second byte

  // It sends the delimiter character 'Z' between the readings. The delimiter
 //is recognized by the Matlab scrip as a divider between the values.
   Serial.print("Z");
   Serial.print(RPM);
   Serial.print("Z");
   Serial.print(OIL);
   Serial.print("Z");
   Serial.print(Diff);
   Serial.print("Z");
   Serial.print(',');
  }
}
////////////////////////////////////////////////////////////////////////////////
////////// END OF THE MASTER SCRIPT ////////////////
////////////////////////////////////////////////////////////////////////////////
```

The Slave Board Script

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////THE SLAVE BOARD SCRIPT//////////////////
////////////////////////////////////////////////////////////////////////////////

// Thanks to the members of the arduino.cc forum for their assistance. The great part
// of this script is the collective effort of the Arduino community.
// This script should be uploaded into the slave board. This scrip is responsible for
// reading the RPM and the oil pressure as well as controlling the accessory relays.

#include <Wire.h> // Header file used for I2C communication

////////////////////////////////////////////////////////////////////////////////
///DECLARATIONS OF RPM RELATED VARIABLES////
////////////////////////////////////////////////////////////////////////////////
// Volatile variables  are used for unstable parameters

extern volatile unsigned long timer0_overflow_count;  // Record the most recent
                                                      // timer ticks
volatile boolean ticks_valid;                         // Indicates a valid reading
volatile unsigned long ticks_per_rev;                 // Number of ticks counted in the
                                                      //current revolution
unsigned long msSinceRPMReading;                      // Number of mSec since the last
                                                      //rpm_sense (used to spot zero RPM)
int lastRPM, peakRPM;                                 // Most recent RPM reading, and highest
                                                      //RPM recorded since last reset

const float __TICKS_TO_RPMS = 15e6;                   // Convert clock ticks to RPM by
                                                      // dividing ticks into this number
                                // The number will change if there are more magnets in an rpm
                                //   (e.g. 2 magnets = 29296.875)

const unsigned int __WAIT_BEFORE_ZERO_RPM = 1000;     // Number of ms to wait
                                                      // for an rpm_sense before assuming RPM = 0.

int INTPIN = 18;            // RPM SENSOR INTERUPT PIN SET FOR 18 (Only for AtMEGA)

unsigned long msSinceSend;     // mSec when the last data was sent to the serial port

const unsigned int __WAIT_BETWEEN_SENDS = 0;          // Number of ms to wait
                                                      // before sending a batch of data.

////////////////////////////////////////////////////////////////////////////////
 /////////////// DECLARATION OF OUTPUTS ////////////////////
////////////////////////////////////////////////////////////////////////////////

int STARTER = 10;                                     // STARTER RELAY PIN
int IGNITION = 11;                                    // IGNITION RELAY PIN
int WARNING_1 = 51;                                   // WARNING 1 LED
int WARNING_2 = 52;                                   // WARNING 2 LED
int RPM_LED = 3;                                // COONECT LED HERE TO SEE IF RPM GETTING PICKED
int eng_press = A0;                                   // ENG OIL PRESS SENSOR
int diff_press1 = A1;                                 // PRESS DIFF SENSOR 1
```

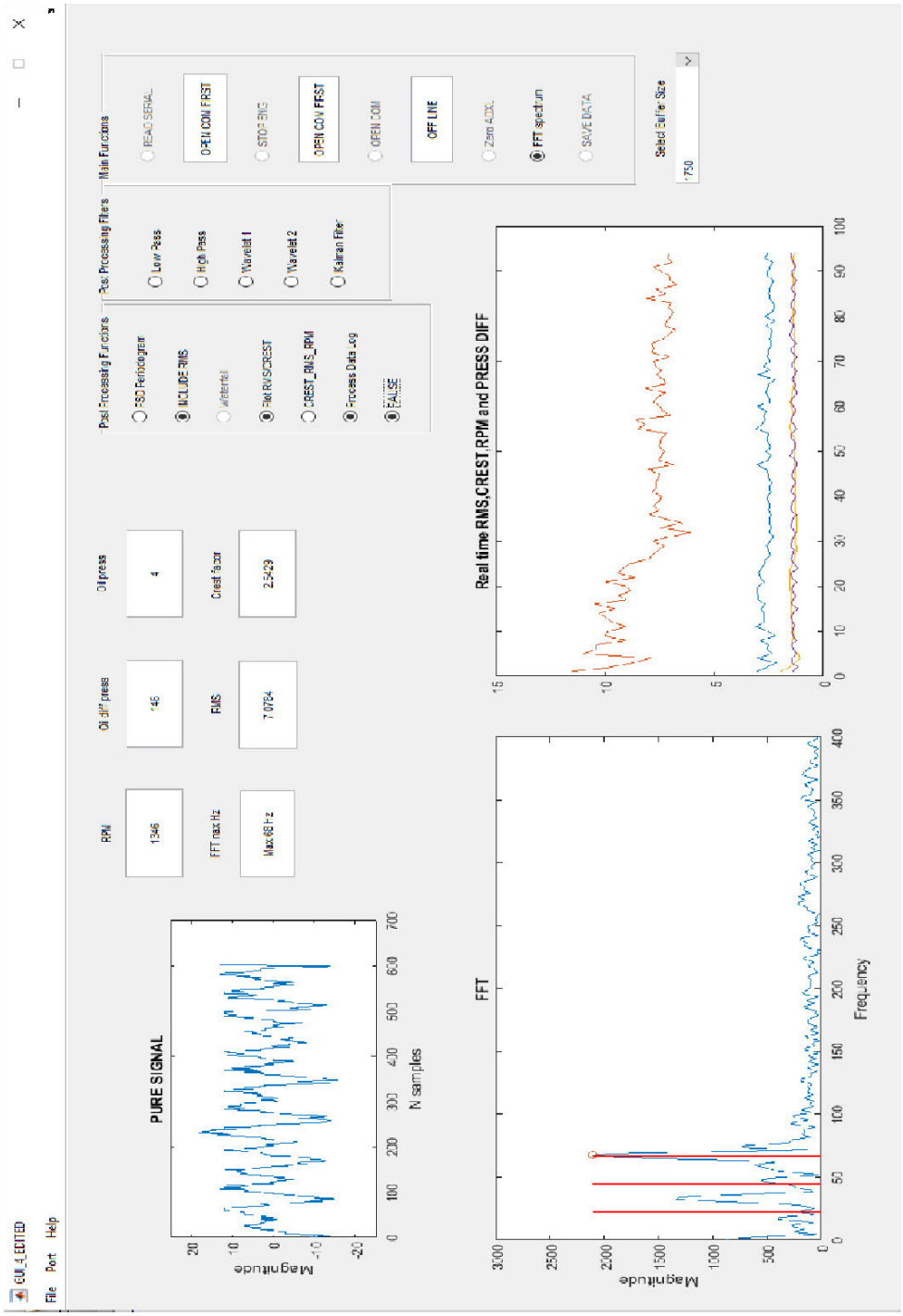
```
int diff_press2 = A2;                    // PRESS DIFF SENSOR 2
float engpress;                  // STORE CONVERTED VALUE OF THE OIL PRESSURE
int engpress_scaled;             // SCALED VALUE OF THE OIL PRESSURE READING
int diff1;
int diff2;
boolean warning1 = false;
boolean warning2 = false;

////////////////////////////////////////////////////////////////////////////////
////////////////////////// INITIAL SETUP//////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

void setup(){

Serial.begin(115200);  // Open serial port at 115200 baud rate
  Wire.begin(8);                         // join i2c bus with address #8
  Wire.onRequest(requestEvent);                 // Send data requested by the master
                                                    // device
  Wire.onReceive(receiveCallback);               // React on data received from the
                                                    // master device
pinMode(IGNITION,OUTPUT);                    // IGNITION RELAY
pinMode(STARTER,OUTPUT);                     // STARTER RELAY
pinMode(RPM_LED,OUTPUT);                     // RPM PICK UP LED
pinMode(WARNING_1,OUTPUT);                   // WARNING LED 1
pinMode(WARNING_2,OUTPUT);                   // WARNING LED 2
digitalWrite(WARNING_1, LOW);                // DISABLE LED 1 BY DEFAULT
digitalWrite(WARNING_2, LOW);                // DISABLE LED 2 BY DEFAULT

 msSinceSend = millis();                     // Data sent counter


 attachInterrupt(digitalPinToInterrupt(INTPIN), rpm_sense, RISING);
// The Hall effect sensor will cause an interrupt on pin2 which will initiate the 'rpm_sense' subroutine.

 msSinceRPMReading = 0;                      // If more than 2000 (i.e. 2 seconds),
                                  // then RPMs can be assumed to be zero (< 15rpm
                                  // at most, with a single magnet, no small IC
                                  // can run that slowly).
 lastRPM = 0;                           // SET Current RPM to zero
 peakRPM = 0;                           // SET Max recorded RPM to zero

}
////////////////////////////////////////////////////////////////////////////////
//////////////////////////END OF INITIAL SETUP//////////////////////
////////////////////////////////////////////////////////////////////////////////


// This subroutine is getting called at the interrupt event. Its comprises the inbuilt
// timer overflow interrupt to provide a stopwatch function. So internally this function
 // counts how many times the timer overflow interrupt occurred since the last RRP
// interrupt was issued.

void rpm_sense()
{
```

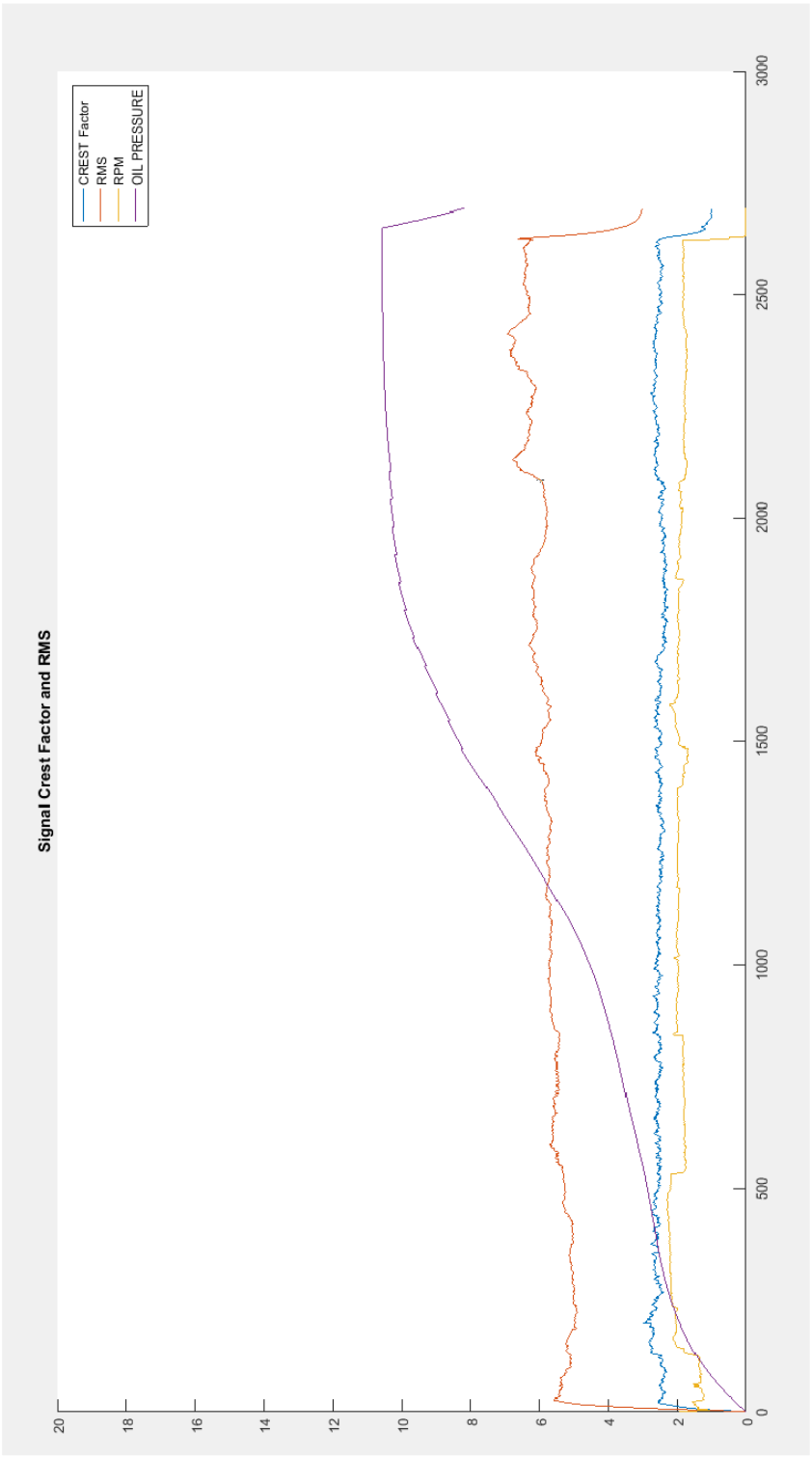
```
 static unsigned long pre_count;              // Last timer0_overflow_count value
 unsigned long ticks_now;

 ticks_now = timer0_overflow_count;           // Read the timer

// To understand the timer registers refer to ATmega 328 specifications
 byte t = TCNT0;
 if ((TIFR0 & _BV(TOV0)) && (t<255))
   ticks_now++;
 ticks_now = (ticks_now << 8) + t;

 if (pre_count == 0) {                        // First time around the loop?
   pre_count = ticks_now;                     // Yes - set the precount, don't use this number.
 }
else {
   ticks_per_rev = ticks_now - pre_count;     // No - calculate the number of ticks...
   pre_count = ticks_now;                     // ...reset the counter...
   ticks_valid = true;                        // ...and flag the change up.
 }
 digitalWrite(RPM_LED,HIGH);                  // COONECT LED HERE TO SEE IF RPM
                                              //GETTING PICKED
}

////////////////////////////////////////////////////////////////////////////////
////////////////// THE MAIN LOOP///////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

void loop(){

engpress=analogRead(eng_press); // READ THE ENGINE OIL PRESSURE

engpress_scaled=(engpress-100)/105; //Scale engine press reading
// The scaling converts the voltage reading into the 0-4 Psi scale

diff1=analogRead(diff_press1);   // READ OIL DIFF SENSOR 1
diff2=analogRead(diff_press2);   // READ OIL DIFF SENSOR 2

if (warning1) digitalWrite(STARTER, HIGH);   // ENGAGE STERTER RELAY
else          digitalWrite(STARTER, LOW);        // KEEP STARTER RELAY DEACTIVATED


digitalWrite(RPM_LED,LOW);                   // KEEP RPM LED LOW BETWEEN INTERRUPTS
unsigned long thisMillis = millis();         // READ THE SYSTEM TIMER (in ms)


 ////////////////////////////// Calculate RPM/////////////////////////////////////////

 if (ticks_valid) {                          // Only come here if we have a valid RPM reading
   unsigned long thisTicks;

   noInterrupts();                           // Disable interrupts while calculating
   thisTicks = ticks_per_rev;
   ticks_valid = false;
   interrupts();                             // Enable interrupts
```

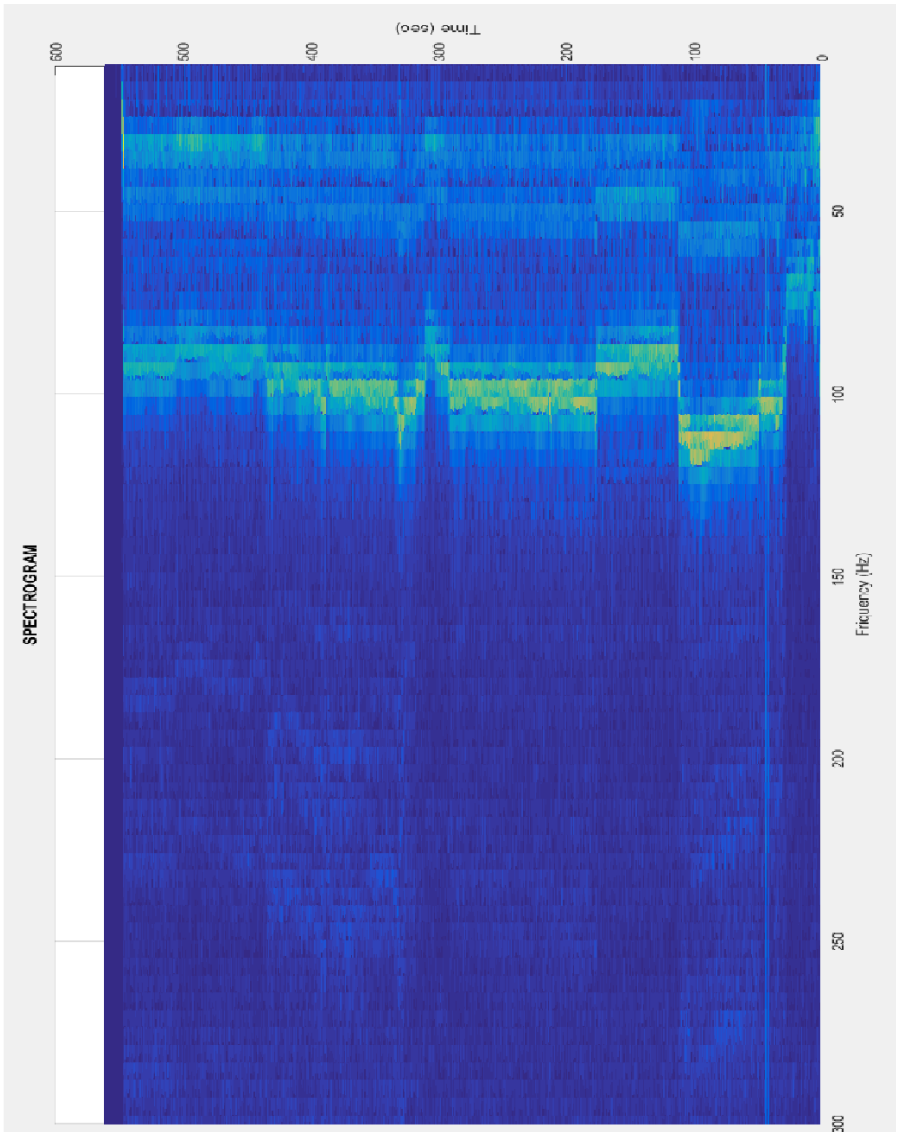
```
  lastRPM = __TICKS_TO_RPMS / thisTicks;      // Convert ticks to RPMs
  ticks_valid = false;                        // Reset the flag.
  msSinceRPMReading = thisMillis;             // Set the time we read the RPM.
  if (lastRPM > peakRPM)
    peakRPM = lastRPM;                        // New peak RPM
 }
else {
                              // No tick this loop - is it more than X seconds since the last one?
  if (thisMillis - msSinceRPMReading > __WAIT_BEFORE_ZERO_RPM) {
    lastRPM = 0;         // At least 2s since last RPM-sense, so assume zero RPMs
    msSinceRPMReading = thisMillis;          // Reset counter
  }
 }

 if (thisMillis < msSinceSend)                // If thisMillis has recycled, reset it
  msSinceSend = millis();

 if (thisMillis - msSinceSend > __WAIT_BETWEEN_SENDS) {

  msSinceSend = millis();                     // Reset the timer
 }
// UNCOMMENT FOR DIRECT OUTPUT TO THE SERIAL PORT
 //Serial.print(engpress);
//Serial.print(',');
//Serial.print(lastRPM);
//Serial.println();
}

////////////////////////////////////////////////////////////////////////////////
////////// Subroutine to send RPM via I2C //////////////////////////
////////////////////////////////////////////////////////////////////////////////

  void requestEvent() {
  int a=lastRPM;
  int b=engpress_scaled;
  int c=diff2-479;   // 479 is zero reading for particular sensor. For different sensors this
                          // value should be determined experimentally
  ////////////////////////////////////////////////////////////////////////////
  //// I2C can only transmit one byte per cycle so we need ////
  ////////////to split long number onto two bytes ////////////////////
  ////////////////////////////////////////////////////////////////////////////
                                  // Split data onto 2 bytes
  byte RPMmyArray[2];    // Two bytes array to store RPM
  byte OILmyArray[2];      // Two bytes array to store the engine oil pressure
  byte DiffmyArray[2];     // Two bytes array to store the debris filter oil pressure

// BYTES MANIPULATION
  RPMmyArray[0] = (a >> 8) & 0xFF;              // Byte shift
  RPMmyArray[1] = a & 0xFF;                     // Combine into two bytes
  OILmyArray[0] = (b >> 8) & 0xFF;              // Byte shift
  OILmyArray[1] = b & 0xFF;                     // Combine into two bytes
  DiffmyArray[0] = (c >> 8) & 0xFF;             // Byte shift
  DiffmyArray[1] = c & 0xFF;                    // Combine into two bytes
```

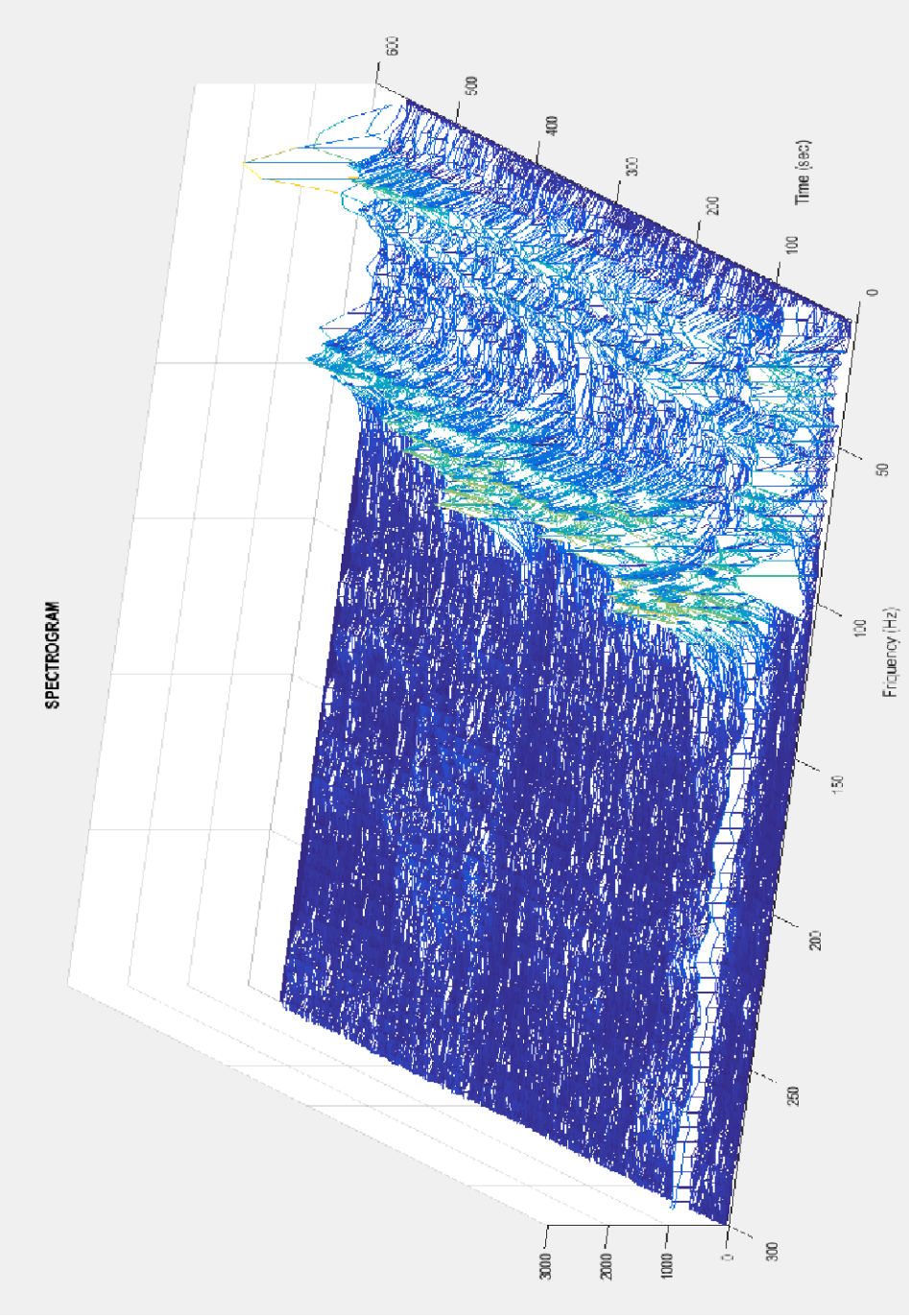
```
  Wire.write(RPMmyArray, 2);                    // Send 2 bytes
  Wire.write(OILmyArray, 2);                    // Send 2 bytes
  Wire.write(DiffmyArray, 2);                   // Send 2 bytes
}
////////////////////////////////////////////////////////////////////////////////
//// Subroutine to read vibration warning sent by MATLAB ////
////////////////////////////////////////////////////////////////////////////////

void receiveCallback(int aCount) {
if(aCount == 1) {                       // If interrupt triggered due to incoming data
byte receivedValue = Wire.read() ;   // Read data from serial port

if (receivedValue >0){            // If incoming integer is greater than zero
warning1 = true;                  // Set the warning flag
}
  else {                            // If received integer equals zero
warning1 = false;                 // Remove the warning flag
     }
   }
}
```

Appendix F - Matlab Output Plots and GUI Interface

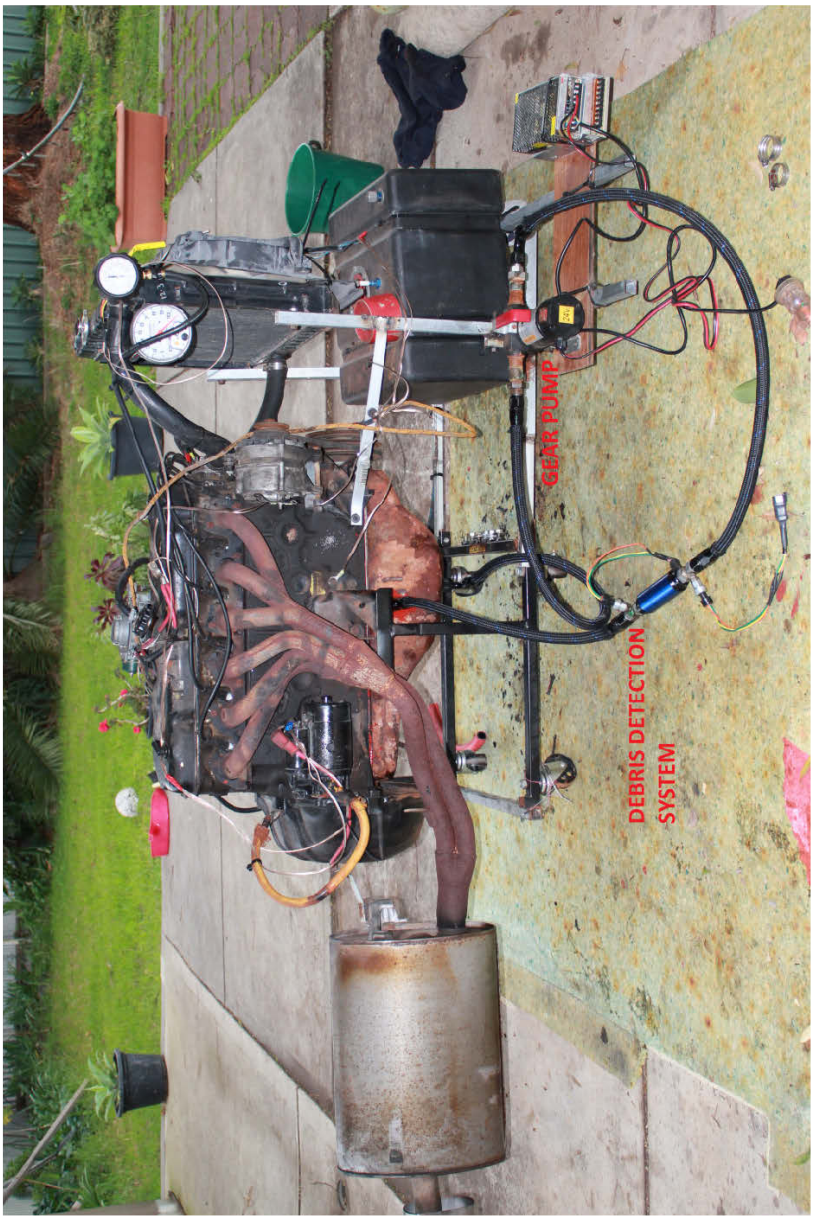



Signal Crest Factor and RMS
CREST Factor
RMS
RPM
OIL PRESSURE
0
500
1000
1500
2000
2500
3000
0
2
4
6
8
10
12
14
16
18
20


SPECTROGRAM
Time (sec)
600
500
400
300
200
100
0
50
100
150
200
250
300
Frecuency (Hz)


SPECTROGRAM
Time (sec)
Frequency (Hz)
600
500
400
300
200
100
0
50
100
150
200
250
300
3000
2000
1000
0

Appendix G - The Test Bench Photos




ACCELEROMETER
SIGNAL AQUISITION
HARDWARE