VIBRATION ANALYSIS & VIBRATING SCREENS: THEORY & PRACTICE

VIBRATION ANALYSIS & VIBRATING SCREENS: THEORY & PRACTICE

By
JAY PARLAR, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2010) McMaster University
(Software Engineering) Hamilton, Ontario

TITLE: VIBRATION ANALYSIS & VIBRATING SCREENS: THEORY & PRACTICE

AUTHOR: Jay Parlar, B.Eng. (McMaster University)

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: x, 185

Abstract

Vibration Analysis (VA) is a key technique used for maintenance and fault detection of vibrating machinery. The purpose of maintenance is to analyze how well the machinery is operating within its target parameters, while fault detection is done to diagnose and locate a fault that might be developing on the machinery.

If we consider $s(n)$ to be the true signal from a rotating system and $e(n)$ to be the additive noise corrupting the signal, then the observed signal is $x(n) = s(n) + e(n)$. If $s(n)$ is composed of a main driving frequency $s_m(n)$ and summed fault frequencies $s_f(n)$, then fault detection is the study of $s_f(n)$. In fault detection, we eliminate $e(n)$ as much as possible so that $s_f(n)$ can be isolated and studied.

This thesis presents a technique based on cross-correlation, utilizing a network of sensors, to eliminate $e(n)$ from the measurements, preserving just the correlated frequency content. This is extended to provide a means of localizing the source of the frequency content, based on the relative strengths of the members of the complete set of cross-correlations between all sensors. This technique has been shown to be able to extract a signal buried by noise, in situations where the traditional FFT fails.

To enable this, a new VA system has been developed. This introduces new wireless vibration sensors as well as a data capture unit capable of providing real-time VA data to technicians. The system can simultaneously capture data from eight sensors, so the data can be used not only for traditional VA techniques, but also in conjunction with the cross-correlation technique described above. This system is now commercially available and in use by dozens of technicians around the world.

Acknowledgements

I would first like to express my sincere thanks to my supervisor Dr. Martin v. Mohrenschildt. His guidance and encouragement throughout the process have been invaluable. Without his technical expertise and dedication to the project, this work would not have been possible. In addition I would like to thank my supervisory committee of Dr. Doug Down and Dr. Spencer Smith for their feedback and comments about the work over the years.

A special thanks to our industry partners at W.S. TYLER for their domain expertise in Vibrating Screens and constant user feedback. In particular, I would like to thank Markus Kopper, Douglas Lima, Dieter Takev and Florian Festge.

The financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC), the McMaster University graduate scholarship, my supervisor and W.S. TYLER is greatly appreciated.

Portions of the work on the Vibration Analysis system were developed in conjunction with Masters students in our group. My thanks to both Sahar Kokaly and Daniel Volante for their contribution and companionship throughout the project.

I would like to express my deepest appreciation to my sisters, Sarai and Melissa Parlar, and my parents Dr. Mahmut Parlar and Irene Parlar for their love, encouragement and support during the thesis and my entire life.

Finally, and most importantly, I would like to express an infinite amount of gratitude for my beautiful wife Mary Ellen Parlar, and our lovely daughter Lyra. From them I have received constant love, support and patience throughout this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the field of rotating machinery, Vibration Analysis (VA) is one of the most widely used techniques for fault detection and maintenance. VA typically involves time-domain discrete sampling of the accelerations of the machinery, followed by both time and frequency domain analysis of the results by expert technicians. As the data is already stored as a set of discrete samples, conversion to and from the frequency domain is an easy and fast operation, thanks to techniques such as the Fast Fourier Transform [1] (FFT).

While much research is currently ongoing to develop new techniques for data analysis of vibration data [2] [3], the fact remains that industry technicians still often only employ basic filtering, FFT and waveform analysis while performing VA in the field. New techniques are required that these technicians will not only be comfortable with, but will also be able to realistically deploy.

Vibration analysis is typically performed for one of two purposes:

- Maintenance/Tuning
- Fault detection

Maintenance includes preventative maintenance: ensuring that the machine is running in a correct state both for the purpose of preventing future faults but also to ensure that it is running optimally/efficiently/etc.

Fault detection is performed to diagnose why a machine is operating incorrectly. It might be that the machine began to make an odd sound, components are wearing faster than expected, or any number of other issues have presented themselves.

To describe this in a more formal way, assume that when measuring a rotating machine, the system presents

- $s(n)$, the true signal of the system
- $e(n)$, the additive noise corrupting the signal $s(n)$

The observed signal $x(n)$ is then

$$x(n) = s(n) + e(n)$$

Vibration analysis is the study of $s(n)$.

Furthermore, consider the separation of $s(n)$ into multiple components: the main driving frequency $s_m(n)$ of the rotating machine, and $s_f(n)$, the summation of any other interesting frequency components present in the machine. These are often caused by faults occurring in the machine.

So the observed signal $x(n)$ can be written as

$$x(n) = [s_m(n) + s_f(n)] + e(n)$$

Maintenance, for the purpose of ensuring optimal and efficient behaviour is the study of $s_m(n)$, while fault detection is the study of $s_f(n)$.

Tools currently used by technicians in the field of VA are often focused on the maintenance aspect. These tools concentrate on the fundamental frequency, employing filters that emphasize the effects of $s_m(n)$ but seriously attenuate *both* $e(n)$ and $s_f(n)$.

When done with tuning and maintenance in mind, there is no problem with this approach. These filters can precisely show the motion of the rotating machine relative to $s_m(n)$ and provide a great deal of valuable information to technicians.

When approached from the fault detection direction, these filters hurt more than help. By attenuating *all* frequency content except that of $s_m(n)$, the interesting effects from $s_f(n)$ are destroyed.

We propose a method of using cross-correlation with multiple sensors attached to a rotating machine to aid in the removal of $e(n)$ from $x(n)$, i.e. removing noise, even very high amplitude noise, while leaving only the interesting frequency content. Not only is this technique useful to VA, but to any system comprised of desired periodic components and undesired uncorrelated noise.

To illustrate the utility of cross-correlation as a means of noise filtering, we will consider a fixd-base rotating machine (though this applies to any system

driven by periodic components). We can simultaneously measure the signals at two different physical locations on the machine, and name them $a(n)$ and $b(n)$.

With two of these signals $a(n)$ and $b(n)$ we can compute the cross correlation $R(a,b)(m)$ as

$$R(a,b)(m) = \sum_{n=-\infty}^{\infty} a(n)b(n+m)$$

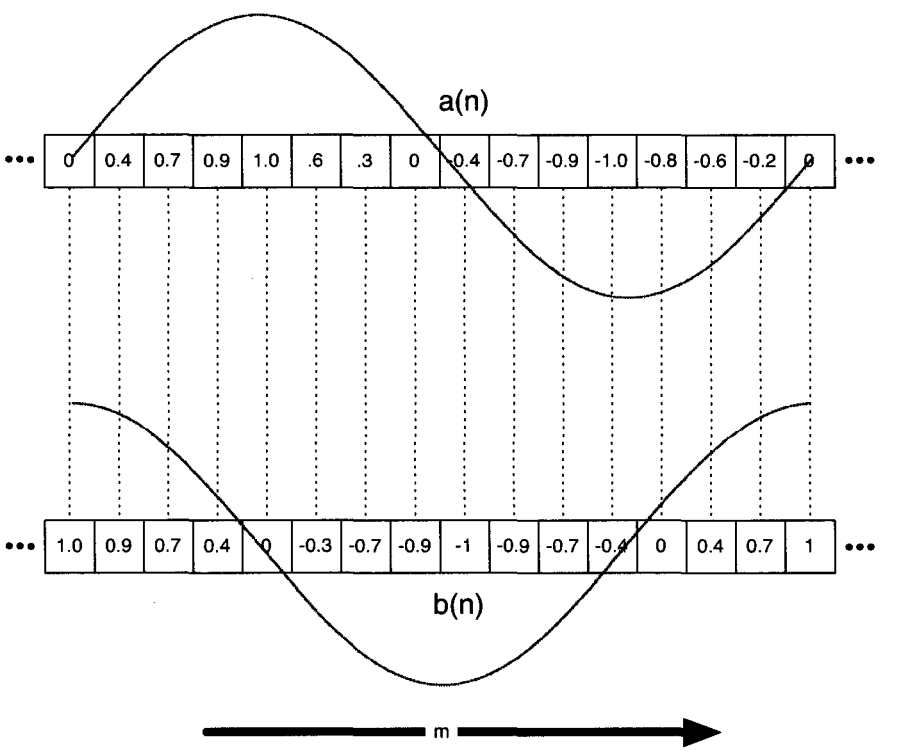

Figure 1.1: Cross Correlation

The index variable m allows computation of the correlation at different *lags* or time points. The more correlated the signals are at some time m, the higher the value of $R(a,b)(m)$.

Figure 1.1 shows an example of this. Signals $a(n)$ and $b(n)$ are identical sinewaves, and for each different lag value m, the signal $b(n)$ is shifted to the

right, and the correlation is re-computed. The particular instance shown in the figure has $b(n)$ shifted one-quarter of a cycle from $a(n)$. As m is increased, the phase shift between the two signals increases, and the correlation drops. As m continues to increase, the signals will move back into phase with each other, increasing the correlation as they go.

The key here is that uncorrelated noise will fall out directly, leaving only the periodic signal $R(a,b)(m)$. Even more importantly, if $a(n)$ and $b(n)$ have the same frequency f, then $R(a,b)(m)$ will *also* have the frequency f. This will be explored in much more depth in Chapter 4.

Using this technique provides a means for eliminating uncorrelated noise from a random process when two random signals exist for that system, providing a means for identifying the true signal.

To apply this technique to VA, an entire hardware and software system had to be developed, which can simultaneously measure multiple points on a rotating machine, providing the signals required to perform cross-correlation.

This VA system had to be capable of doing at least what is already possible with existing (single measurement point) industrial VA systems, to make it acceptable as a replacement for older systems, but also had to be capable of measuring multiple points simultaneously. While theoretically a hardwired system would have worked here, for safety and technician-convenience reasons a wireless system was required, adding an extra layer of complexity to the task.

Such a system was developed as part of this work, and is now commercially available and in use by VA technicians across the globe.

The hardware and software of the system were developed in a generic way so as to provide a platform for future VA work and research. Both are extensible and existing components can easily be modified or swapped out. In fact, it is already being extended as part of a research project to develop fixed-installation continuous monitoring and control for a certain class of vibrating machinery.

Many important lessons were learnt in the development of the hardware and software, important to other researchers wishing to embark on similar tasks. Lessons learned, pitfalls encountered and problems solved will all be presented here.

The hardware and software system was developed as a specific case study for VA of vibrating screens (a mechanical device which will be introduced later), but most of the work was generic enough as to be useful to any VA researcher.

1.1 Motivation, Goals and Objectives

Current VA systems employed in industry that we have experience with are limited to a single accelerometer in communication with a Data Acquisition Unit (DAU). Performing VA on an entire system consists of measuring the interesting points of the system, individually, and then analyzing the data using waveforms, acceleration orbits and the frequency spectrum. The accelerometers are typically attached to the DAU via a wire, impeding a technician's ability to move around during data acquisition, and causing safety concerns for those technicians.

The overarching goal of this project was to create new methods for performing VA and interpreting the results, methods that would be feasible for actual implementation.

A common problem in many fields is that of frequency identification within a captured signal. In a VA setting, this would involve identifying interesting frequencies from recorded accelerometer data. The presence of different frequencies can often indicate certain faults or malfunctions in the rotating machinery. These frequencies can be buried under noise making them very difficult to identify and extract. A technique for frequency identification and noise removal using a network of sensors and cross-correlation is introduced in this thesis, applicable to many situations where multiple recordings of a single system are available. This is extended to the particular case of VA, and in particular, to the topic of fault location detection, the process of locating the physical location on a piece of rotating machinery from which a fault is emanating.

This thesis also explores the requirements that would be necessary to build an advanced VA system, identifying potential pitfalls and needed components. In particular, these systems should be built in a manner such that they present themselves as a generic framework for VA; not just limited to the capabilities of the system as implemented, but with an eye to the future, to make integration of later developments in VA as simple as possible. The system should be capable of wirelessly recording multiple sensor points simultaneously. This is required to increase safety for the technicians, increase time correlation between the measured signals, and to simply reduce the amount of time it takes to perform a full data capture over all interesting points.

To solidify these requirements, a full VA system has been built as a case study. This system is now in use by VA technicians around the world. Important details and insights into the design and implementation of this system are presented, as well as lessons learnt.

The system itself is comprised of both hardware and software components,

each of which will be described. It allows for a network of sensors to simultaneously monitor a vibrating system, as opposed to the single-sensor systems found in industry, and in the literature. The component costs of the hardware platform were kept in mind at all times, again to promote a final solution that would be feasible for real-world use. The software platform was designed with a Model-View-Controller (MVC) [4] architecture, with a short-term goal of implementing two drastically different user interfaces (the View component), as well as providing a means to abstract-out the data input methods so the sources of data can be easily swapped (real-time sensor data vs previously recorded data, for example).

1.2 Novelty of the Research Presented in this Thesis

This thesis provides two primary contributions. The first is the noise removal and frequency identification/localization technique based on cross-correlation of multiple sensors monitoring a single system. This technique should be applicable to any system where signal noise is burying potentially interesting frequency content. A specific use case applied to VA is presented, with particular focus on vibrating screens. The technique is used to aid in vibration location detection on these screens.

To be able to perform this technique on real vibrating machines, and test its efficacy, a new type of VA system was required, one capable of simultaneously acquiring data from multiple locations on a vibrating machine.

As such, the second contribution is the development of both the hardware and software for a multi-sensor wireless VA system. This system is capable of performing many traditional VA techniques, as well as the vibration location detection technique described above. The main design criteria are presented, as well as discussion of pitfalls and issues encountered during the system's development. The system was designed in such a way as to provide a platform for future VA research and developments. For the purposes of this research, particular emphasis was placed on abstractions around key components within the system, generalizing the interfaces such that replacement of those components would be relatively simple. Researchers wishing to develop similar systems will find many useful results from this work. Work has already begun on taking the research and systems presented here and extending them towards a fixed-installation continuous monitoring VA system. Through the work on these two primary contributions, two secondary contributions presented themselves.

The first is a description of an issue surrounding high-speed data transfers

within a Bluetooth network, and the various hardware and software components in the system that play a part in these issues. Our solution, involving handshaking protocols, buffering schemes and atomic memory access, is described in detail. While the system here makes use of Bluetooth, the results are generally applicable to any wireless sensor network.

The second is an analysis of another wireless network issue we encountered. This is related to the synchronization of sensor nodes, and the ways that less-than-ideal synchronization techniques affect the system. The details of this and lessons learnt are applicable to anyone building a wireless sensor network. Solutions are also presented, as well as descriptions as to how the networking issue affects the current and proposed VA techniques.

1.3 Problem Statement

The primary problem dealt with in this work is:

> Aiding technicians in the field of vibration analysis, improving their tools and techniques for both maintenance and fault detection of rotating machinery.

As such, the original goals of this thesis can be summarized as follows:

1. Develop algorithms and techniques for improved instantaneous fault detection in vibrating systems
2. Design and implement a multi-sensor wireless VA system, helping to identify missing requirements and pitfalls that designers of similar systems might encounter

While the first goal is listed as "fault detection", in actuality the developed process is more general than that. The algorithm and tool developed aid a technician in localizing vibration sources on a running machine. These sources may or may not be directly related to a present or developing fault, that is left up to the technician to determine.

The end result of both of these goals is to aid technicians in the process of VA, so the tools must not only be applicable in a laboratory setting but also usable in the field. To this end, the developed system has undergone extensive testing and use by industry technicians in the field of VA, resulting in an iterative feedback

cycle where comments and issues were taken in, addressed, and improved systems were sent out to the users.

The sensors should be capable of withstanding the harsh environments that much VA takes place in, must take into account potential issues such as signal strength, signal range, available bandwidth, power consumption, etc.

The entire system not only has to work in the ideal case, but also in less than ideal circumstances. Can it be used in low-light scenarios? How does it behave if a battery dies during operation? Does it take a long time to setup a test? Will the system actually be usable if dozens of rotating machines need to be analyzed in a very short period of time. These are questions that are often ignored in systems meant for the lab, but were deemed vitally important in this work.

1.4 Vibration Analysis System

As mentioned, one of the contributions of this thesis is the design and implementation of a wireless VA system. The work here will attempt to present acceptable requirements for *any* wireless VA system, not just the one developed for this project. In addition, the requirements and design, as much as possible, will be presented in such a way as to be useful to the development of any wireless sensor system. Many of the components and needs of a VA system are identical to a wireless system performing some other form of measurement.

The high-level goals for such a system include:

- Wireless sensors
- Multiple sensors running simultaneously
- A portable DAU which coordinates with and controls the sensors
- Data Acquisition unit which can perform multiple types of computation and analysis

A high-level system component diagram is shown in figure 1.2. This will be referred to throughout the thesis, to give the reader context for given sections.

This diagram illustrates the two major components, the wireless sensor, and the DAU. In actuality, the real system is run with multiple sensors simultaneously, but for simplicity in the image, only one is shown.

Within each wireless sensor are five generic components. The analog sensor is the first. In the VA system presented here, this is an accelerometer, but for other

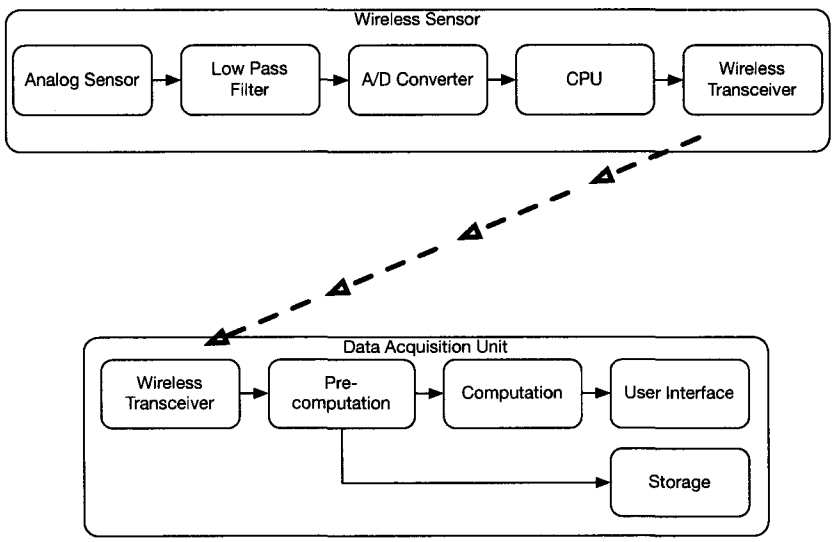

Figure 1.2: System Structure

types of systems, it could be anything. Temperature sensors, pressure sensors, light sensors, etc.

The next part, the low pass filter, is present simply to smooth out the values coming from the analog sensor. Some systems might prefer to remove this component, depending on need.

Next is the A/D converter. This is required to convert the analog values from the sensor into digital values that can be used and transmitted. Depending on the particular hardware selected, the sensor might incorporate the A/D converter internally. In our system it did not, and there are enough interesting aspects to the A/D converter to warrant showing it here (section 5.2.5).

This is followed by the CPU, responsible for collecting the values from the A/D converter, performing any necessary pre-transmit computations, packing up the values for transit, and passing them off to the wireless transceiver. The wireless transceiver, which could be based on any number of wireless technologies, will typically perform all necessary networking operations and perform the actual act of sending the values from the CPU over the air to the DAU.

Within the DAU, a matching wireless transceiver is the first component. The values received here are passed to the pre-computation block.

For our system, pre-computation includes unpacking the transmitted values (dependent on the particular packet format chosen), as well as performing DC

filtering. DC filtering simply removes the constant gravity component from the recorded accelerometer values. The technicians never see the pre-DC-filtered values, so this is considered part of the pre-computation.

The computation block performs further filtering (Butterworth), does FFT computations, RPM calculations, stroke calculations, etc. The particular computations here are very dependent on the field, but in general comprise all mathematical computations that will interest the user.

Values from the computation block are simultaneously sent to the storage and UI blocks. Storage is simply permanent storage of the values, for later analysis. The UI block is responsible for taking the computed values and displaying them on screen in a manner that is useful to the end-user.

Our hope is that this structure is general enough to be useful to anyone designing their own wireless sensor system. As has been presented here, it is overly general, but further specifics will be given throughout the course of this work.

No other VA system with this full set of features was known to be available at the time of this research. The eventual hardware *and* software of the wireless sensors was designed and implemented from the ground up, while the DAU itself is a manufactured component with completely custom VA software running on it.

The selected DAU is shown in figure 1.3 and the final production sensors that we manufactured are shown in figure 1.4

This system had to be capable of performing traditional forms of VA, including RPM identification, filtering and orbit plots, but also able to be used for the new vibration location detection technique introduced in this thesis. Details of the more traditional forms will be given in Chapter 2, while specifics of the vibration location detection are presented in Chapter 4.

1.5 Overview of the Thesis

Chapter 2 provides background details for the field of vibration analysis. This includes different types of analysis as well as mathematical techniques often employed. Vibrating screens are used as part of a case-study for this thesis, so information on these screens is also presented here.

Chapter 3 provides a literature review focusing on vibration analysis, noise removal and fault detection.

Chapter 4 presents our theory for using cross-correlation as an ideal filter. Cross-correlation is introduced, as is our definition and criteria for an "ideal filter". A formal distinction is made between noise and features of a system, and finally a

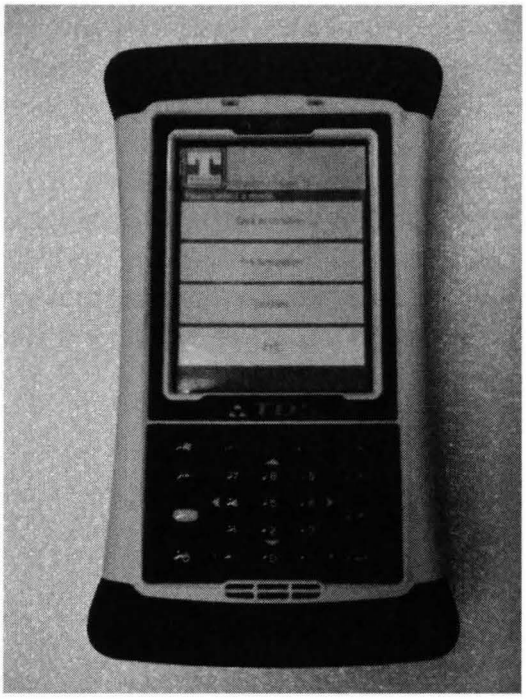

Figure 1.3: Data Acquisition Unit

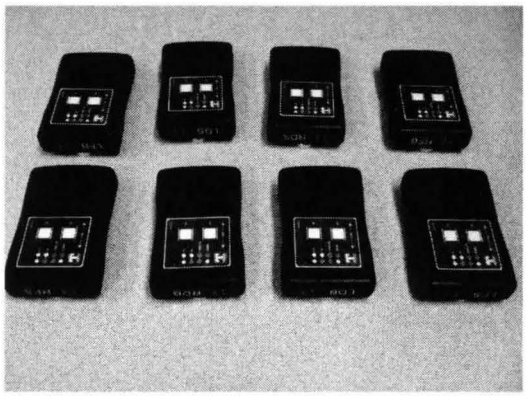

Figure 1.4: Sensors

technique is presented for using cross-correlation to identify the possible location of vibration sources on a vibrating system.

Chapter 5 details the design and implementation of the wireless sensors built

for this thesis. Pitfalls encountered and solutions discovered are presented, with the goal of aiding other researchers hoping to develop similar systems. This chapter covers both the hardware and software that comprise the sensors.

Chapter 6 covers the design and implementation of both the software for the DAU as well as the Vibration Location Detection Tool.

Chapter 7 describes in detail a particular network issue encountered in our system. While this issue does not actually affect the work done in this thesis, it is possibly important to anyone hoping to do other forms of vibration analysis with the system. It is also relevant to any wireless sensor system, so the faults and possible solutions are generally applicable.

Chapter 2

Background of Vibration Analysis

In the field of VA, there are a few techniques widely used in industry, including: orbit analysis, waveform analysis, FFT analysis and filtering. Any tool proposed as a candidate for use by technicians in VA should, at a bare minimum, be able to implement these techniques, as well as any new techniques it might bring to the table.

The common goal of most of these techniques is to analyze frequency-related information in the presence of noise, for the purpose of understanding how a vibrating system is behaving.

Novices in VA often think there is a single "correct" way to process and interpret vibration signals [5]. In reality though, the nature of the problem plays a significant role. Some problems can be solved with a simple glance at the vibration waveforms, while others require more complicated filtering, frequency analysis, time averaging, etc.

For example, with a steady vibrating system like an engine or a pump, the average over many cycles is sometimes important, while at other times it may be the difference between cycles that matters. The approaches to these are different. Similarly, the steady vibrations of a gearbox relay one kind of information, but it is unsteady vibrations that give clues to tooth damage.

This chapter will introduce a selection of techniques. These include the graphical techniques of orbit, waveform and FFT analysis, as well as FFT-based RPM calculation techniques and the DC and Butterworth filters. Orbit and waveform analysis operate in the time-domain, while FFT analysis is done in the frequency domain. The DC filter is used to remove DC (constant) components from a signal, while the Butterworth filter is often used as a bandpass filter, necessary when isolating a particular frequency component. While one purpose of the DC and

Butterworth filters is to "clean" the data so it may be better used with the graphical techniques, other calculations are often based on the outputs of these filters.

The basic mathematical components of these filters will be described, and their effects and characteristics illustrated.

Especially when dealing with filtering techniques, the point from Chapter 1 must be kept in mind: VA can be distinguished as for maintenance/tuning or for fault detection. For example, the Butterworth filter described later in section 2.6 is employed as a bandpass filter, completely attenuating all frequency content except for that around a desired centre frequency. For the purpose of maintenance and tuning, this is ideal. Centre the filter around the operating frequency of the machinery and analyze the output. A visual inspection will clearly illustrate whether or not the system is operating as it should be.

However, this particular filtered output cannot be used for fault detection. If we assume that faults show themselves as frequency components in a measured signal, then a bandpass filter, which attenuates those very frequencies, will be useless.

So different techniques must be used when VA is approached as a fault detection activity. The technique we introduce in Chapter 4 consists of a particular kind of filter that strictly highlights frequency components, both the main operating frequency and any other components that might be present. And while one cannot directly say that those frequencies are indicative of a fault, they do provide a first-course of investigative action for a technician performing VA on a machine.

In terms of the measurement source for the techniques illustrated here, accelerometers will be assumed. In the past, much VA was done using microphones as the primary sensor, and performing frequency analysis on this input. In more recent years, accelerometers have become quite popular for VA, and the techniques presented here assume three-axis accelerometers.

The introduction to the thesis briefly describes a case study in which a hardware/software system for performing VA on vibrating screens was developed. The system implements all of these traditional techniques, as well as some new ones. To provide the reader with a concrete context in which to understand the techniques described here, the techniques will be presented using real data from vibrating screens. This is only done to aid understanding, as the techniques themselves are applicable to almost any fixed-base rotating system

As such, this chapter will begin with a background summary of vibrating screens.

2.1 Vibrating Screens

A vibrating screen is a fixed-base rotating mechanical device used to sort and classify aggregates. Depending on the characteristics of the desired aggregates, machines of various sizes and screens of varying fineness will be used.

A common use case for vibrating screens is in the field of mining. Materials from a mining site are fed onto a mesh screen attached to a vibrating machine, which provides an excitation force, causing aggregate of the right size to filter through the screen. These machines can be capable of sorting through dozens of tonnes of material per hour. Some machines generate up to 7Gs of acceleration, running in a circular, elliptical or linear motions at close to 1000RPM.

Traditionally, to perform VA on this class of machines, a VA system consisting of a single accelerometer and a large data capture station was used. A technician would record accelerometer data from one point on a machine, move to a second point, record data, move to a third point, etc. Depending on the class of machine, there could be up to eight interesting points for data capture. As the system could only record one point at a time, the correlation between the recorded readings was potentially greatly reduced.

The accelerometer measures accelerations in the machine, measured in G ($9.8 m/s^2$). The system samples the accelerometer, creating a Time-vs-G data set. These accelerometers can measure three axes of motion simultaneously, X, Y and Z.

Figure 2.1 shows the side and top views of a typical vibrating screen. The side view is what you would see if you were looking at the side of the machine, and the top view would be seen if standing above the machine and looking down. Material enters the screen at the Feed end, and anything that does not pass through the screen comes out again at the discharge end. Most vibrating screens are mounted at an angle, with the one in the figure being 20°. This is simply so gravity will assist in moving the aggregate over the screen.

While the axes could be interpreted in any way, for historical reasons X is considered to be along the length of the machine, parallel with the material flow. Together X and Y define the main motion of the machine. The Z axis is perpendicular to the main motion of the machine. Theoretically there should be no motion at all along the Z axis, but realistically this is never the case.

At a typical mining site, it is not unusual for other large machinery to be in operation at the same time as the vibrating screens, in fact, it is common. These machines (for instance, dump trucks capable of carrying 400 tonnes) are often so large and powerful that they can introduce motion into the vibrating screens, even

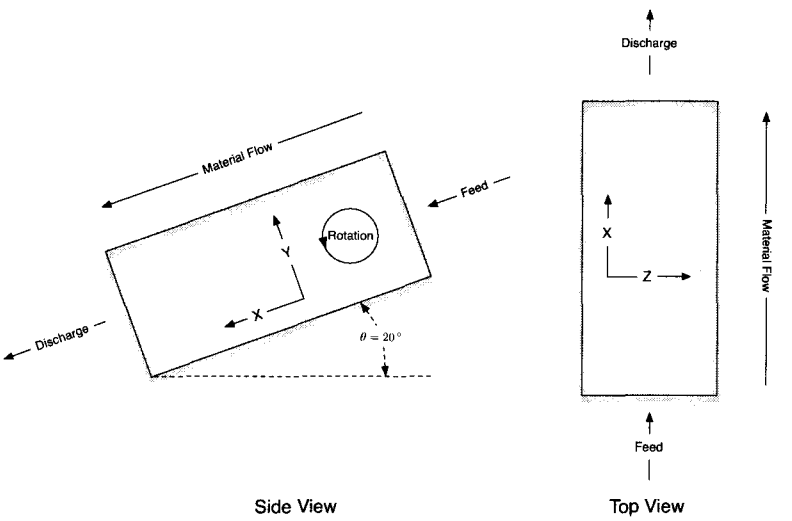

Figure 2.1: Vibrating Screen - Side and Top Views

at quite a distance. If some of this other machinery happened to be reacting in one way at time point A, and a completely different way at time point B, the recorded data of the vibrating screen might differ if the recordings happened to be taken at separate time points A and B. These machines tends to be large enough that it is not unheard of for one to induce harmonics in another.

From a purely practical point of view, a system that could simultaneously record multiple points on a vibrating screen not only reduces the amount of time it takes to gather the readings, but also provides better correlated data. This desire, and the system required to implement it provided an opportunity to try to improve the state of the art of VA.

The screens under analysis for this case study will make best use of four or eight sensors, depending on whether or not the screens are two or four bearing.

The naming scheme of the key positions is illustrated in figure 2.2. This is the same top view of a screen shown in figure 2.1. The first letter of each location, "L" or "R" is for "left" and "right". The left side of a screen is always interpreted from standing behind the machine looking at the feed end. The second letter, "D" or "F" is for "discharge" or "feed". The third letter, "B" or "S" is for "body" or "side-arm", and simply illustrates whether or not the sensor is placed directly on the body of the screen or mounted to one of the side-arms. The particulars of this are not important to the work presented here, but are given simply for clarity

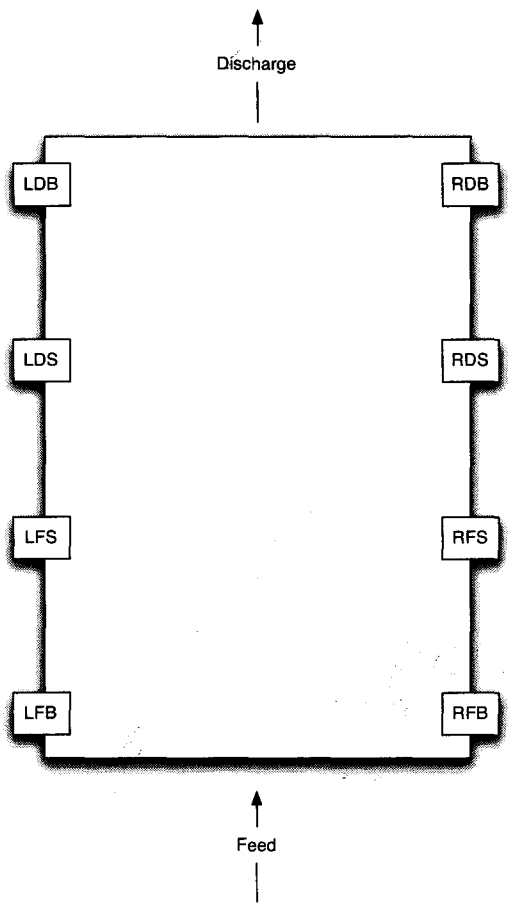

Figure 2.2: Screen Location Names

2.2 Orbit Analysis

As mentioned above, the accelerometers measure three axes simultaneously. The X axis and the Y axis are interpreted as the main axes of motion, the axes on which the circular or linear motion of the vibrating screen occurs. The Z axis is always interpreted as the axis perpendicular to this main motion.

Orbit Analysis is the plotting of the axes against each other and interpreting the results. Namely, X vs. Y, Y vs. Z and X vs. Z are all plotted.

Plotting these axes against each other essentially allows one to view the motion of the machine in two-dimensions. A technician will know what the motion

of a machine *should* be, and the plots allow for this to be verified. For example, a vibrating screen with a circular throw (i.e. circular motion) should give a circle when X vs. Y is plotted, while both X vs. Z and Y vs. Z should be straight lines. Ideally, the Z-related plots should not only be a straight line, but also a perfectly vertical straight line. Any skew outside of perfect verticality implies that the vibrating screen is experiencing some kind of off-centre motion, which is typically undesired.

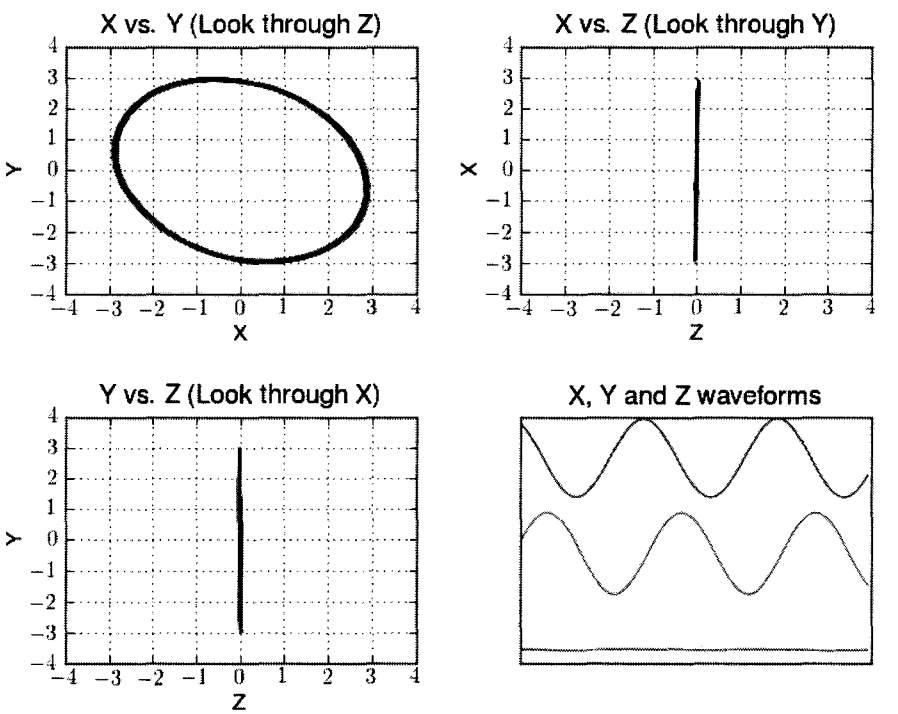

Figure 2.3: Orbit Plots

Figure 2.3 shows all three orbit configurations, as well as the raw X, Y and Z waveforms responsible for these orbits.

To interpret these in terms of motion of the machine, the viewer must imagine positioning themselves looking through the unused axis of a particular plot.

For example, the X vs. Y plot shows the motion of the machine when standing at the side of the machine, looking through the (unused) Z axis. X vs. Z is the mo-

tion when "hovering" directly above the vibrating screen, looking down through the Y axis.

2.3 Waveform Analysis

Of these techniques for VA, Waveform Analysis is the most basic. It is simply a plot of the raw data acquired from a sensor, plotting G-force against time. For a vibrating screen, or any rotating machinery, the main motion should be periodic and harmonic, so when plotted it should be a simple sine wave. Figure 2.4 shows data we captured from a vibrating screen with a loose deck casting, and figure 2.5 shows the same screen with the deck casting repaired.

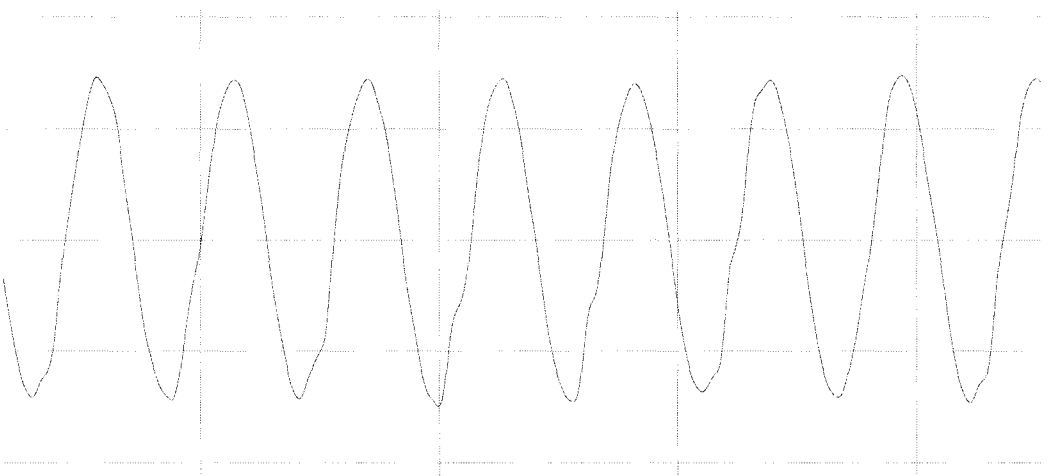

Figure 2.4: Loose Deck Casting Waveform

While Waveform Analysis provides a quick method for a technician to see if a machine is running properly, it suffers when trying to determine why a machine is *not* running properly. A malformed sine wave shows that *something* is wrong, but generally provides no information as to what the problem might be. Moreover, for different definitions of "malformed", it might be that nothing at all is wrong with the system, it just happens to show small jitters in the waveforms even when operating ideally.

The figure above shows how waveforms can be used to see low-frequency problems, but as soon as higher frequency components are present, the waveform is much less useful.

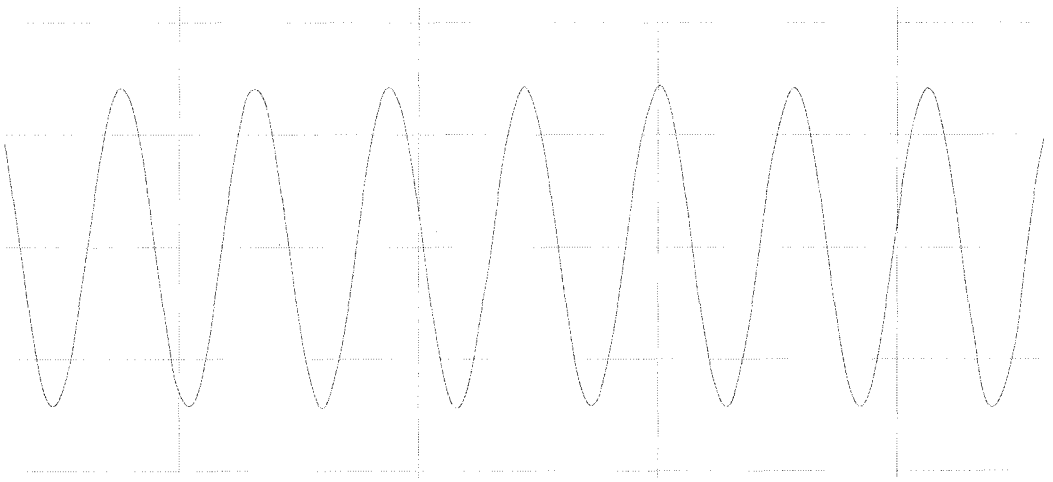

Figure 2.5: Repaired Deck Casting Waveform

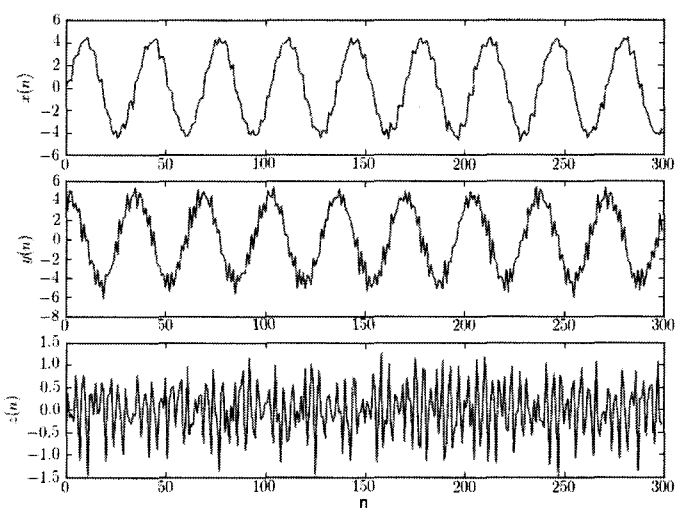

Figure 2.6: Failing Bearing Waveform; X, Y and Z Axes

Figure 2.6 shows a waveform from a running vibrating screen where a bearing was in the initial stages of a failure. Bearing faults usually show themselves through high frequency components in the measured signal, and depending on the

way the sensor was placed on this screen, that harmonic component is usually seen most strongly through the Z axis.

With the high frequencies in the Z axis, it is nearly impossible to use the waveform, other that to say that something high-frequency is occurring. The primary frequency component is still visible through the X and Y axis, but even those show a high degree of high frequency harmonics on them.

The FFT provides a more realistic way to analyze high frequency components of a signal.

2.4 FFT Analysis

Probably the most powerful VA technique currently employed by technicians in the field is that of FFT Analysis. Given time domain measurements, such as those from the last section, the FFT is able to analyze the frequency components that comprise the time domain signal, showing relative amplitudes of the various frequencies present in the signal.

Returning to the failing bearing example introduced in the last section, figure 2.7 shows the same measurements, but in the frequency domain. Notice in the Z axis the components present at 114Hz, 144Hz and 228Hz.

The bearings used in vibrating screens usually come with fault tables, showing which measured frequencies are associated with a particular type of bearing fault. For the system under analysis, 114Hz fell perfectly into the range of one of the known types, indicating that a particular fault is probably occurring.

The interpretation of meaning behind the presence of different frequency components is highly dependent on the machine under study. Some knowledge of the behaviour of the machine under different conditions is thus a pre-condition to successfully interpreting an FFT analysis. This can lead to situations in which an harmonic is present in a system that has never before been seen in that type of system, necessitating further study and analysis.

As another example, figures 2.8 and 2.9 show the same loose casting screen from the previous section. In both figures, the fundamental 14Hz frequency is clearly visible, but a great deal of high-frequency content is present with the loose deck casting. After the deck casting is repaired, most of the high-frequency content simply disappears. Technicians thus will often use the presence of high frequency content as a possible indicator that a fault is developing on a machine.

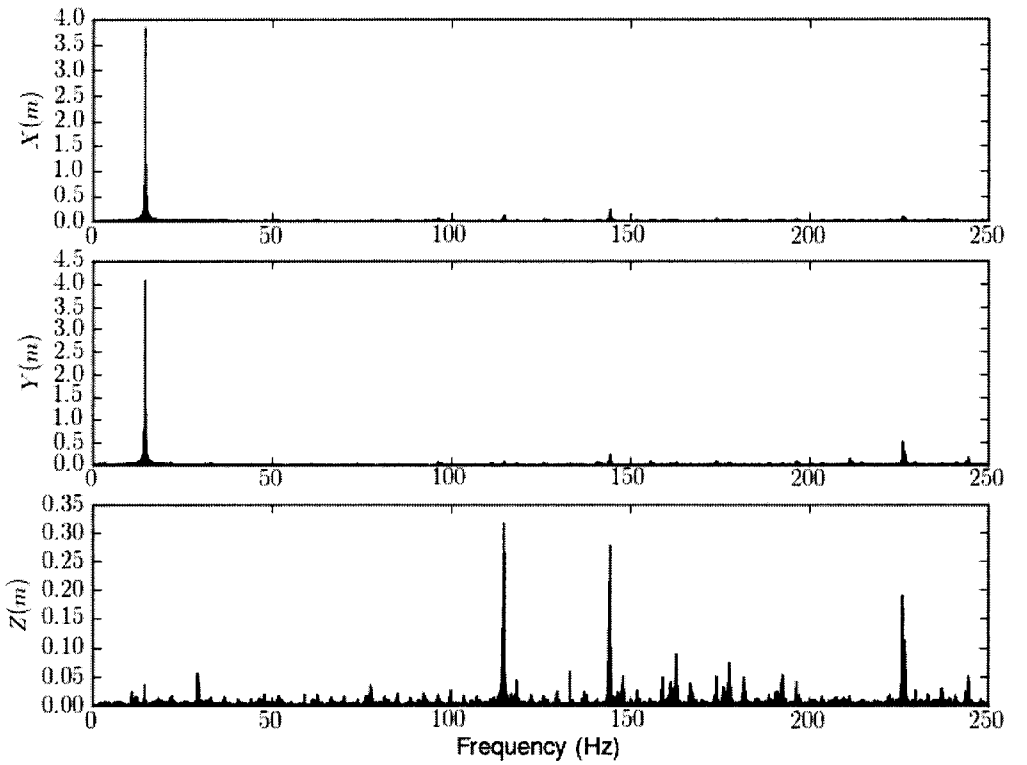

Figure 2.7: Failing Bearing FFT; X, Y and Z Axes

2.5 DC Filter

The DC filters, often called DC blockers [6] are used to remove the constant component from a signal. They are implemented with a small recursive filter as

$$y(n) = x(n) - x(n-1) + Ry(n-1)$$

where R is typically somewhere between 0.9 and 1. Smaller R values allow for faster tracking of "wandering dc levels" [6]. These smaller values will result in greater low-frequency attenuation, so the value should be chosen in accordance with the situation.

With fixed-base rotating machinery, the DC value is the gravity component

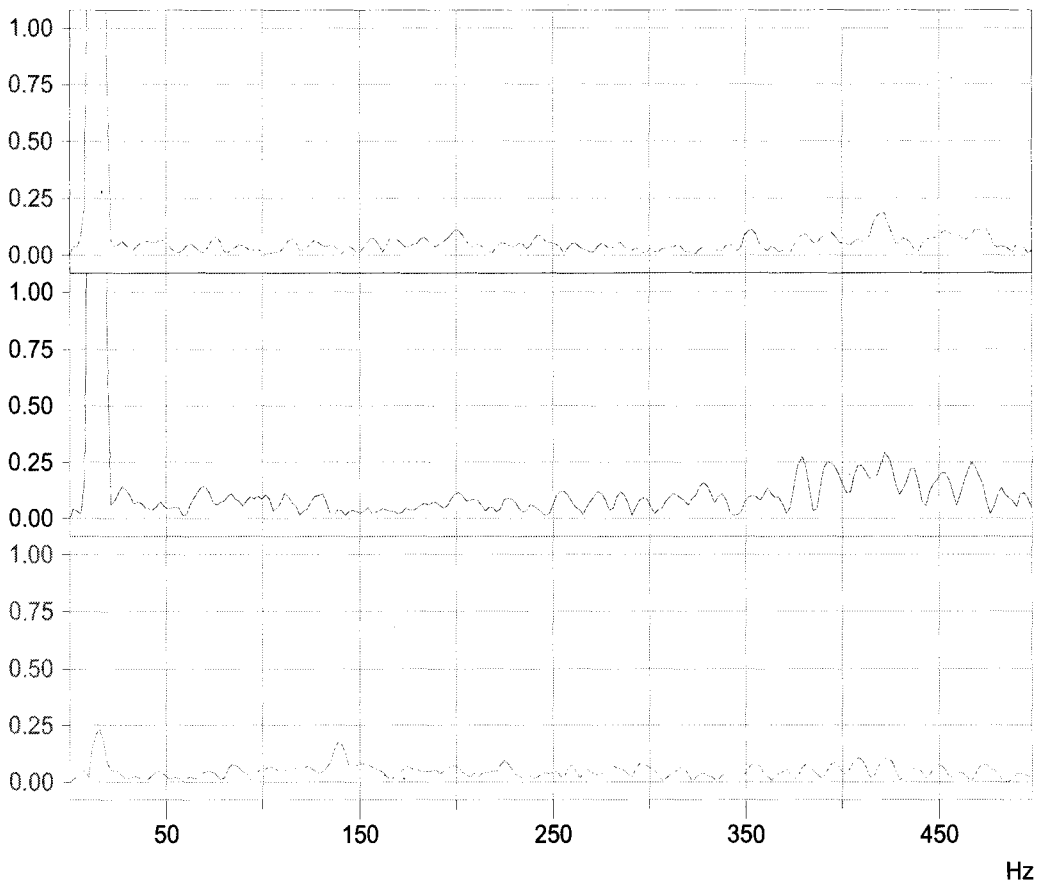

Figure 2.8: Loose Deck Casting FFT

measured by the accelerometers. Since this DC component should not wander (i.e. gravity will not change), a relatively high value of 0.98 can be used. This minimizes low-frequency attenuation as the ability to quickly track wandering DC levels is unnecessary.

If the vibrating screens were installed exactly perpendicular to gravity then the gravity component could be subtracted out from the Y axis without requiring a DC filter, but this is generally not the case. Typical screens are mounted at some angle (as shown in figure 2.1), so gravity will have some effect on both the X and Y axes. Individual DC filters on the X and Y axis are thus used to remove whatever gravity effect is present.

Figure 2.10 shows the results of passing simulated input through a DC fil-

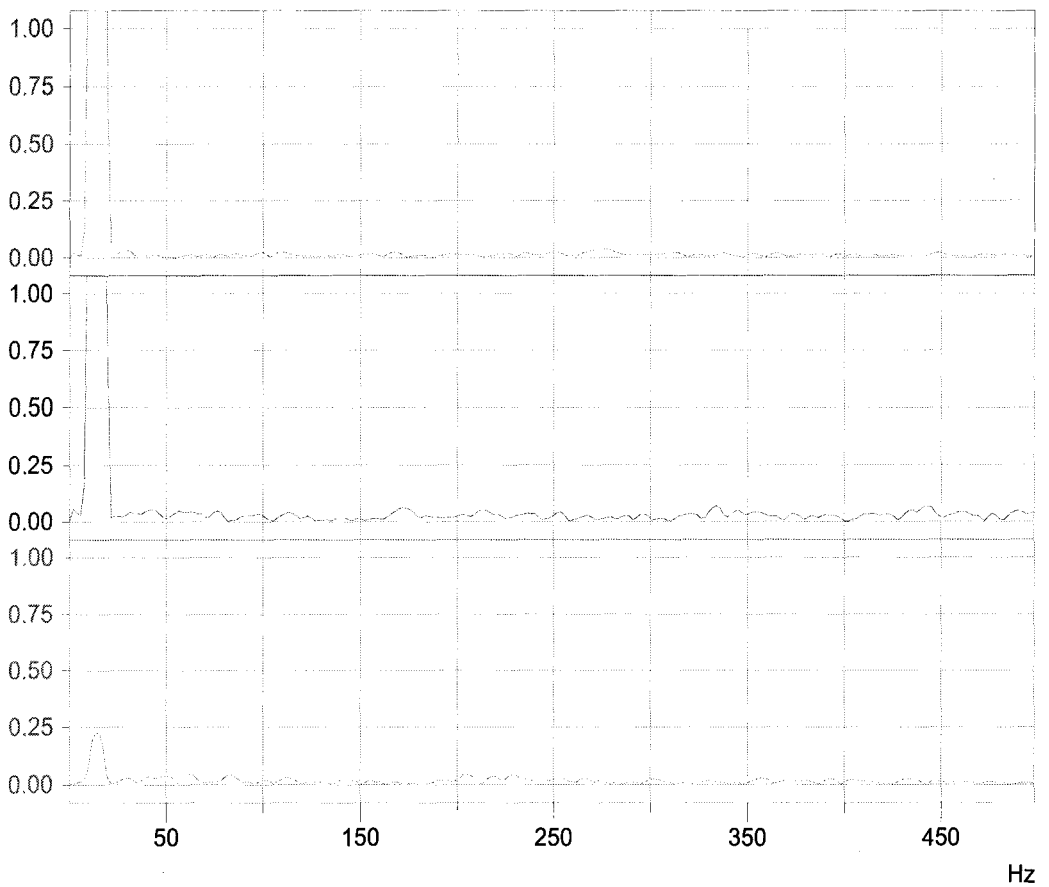

Figure 2.9: Repaired Deck Casting FFT

ter. Two separate input channels $x(n)$ and $y(n)$ are simulated, both 15Hz signals sampled at 500Hz (one sine and one cosine), which roughly matches sampling a typical vibrating screen. A constant value of 2 is added to each system to provide the DC filter a component to remove.

While a typical system will not actually have a constant of 2 on each of the X and Y axes, it does not matter. The DC filter does not care what the magnitude of the constant component is, only that there *is* a constant component.

Two separate DC filters, DC_x and DC_y were used, one for each channel, and the two simulation channels were passed through them, resulting in $x_{dc}(n)$ and $y_{dc}(n)$. The DC filters perfectly removed the constant components without affecting the

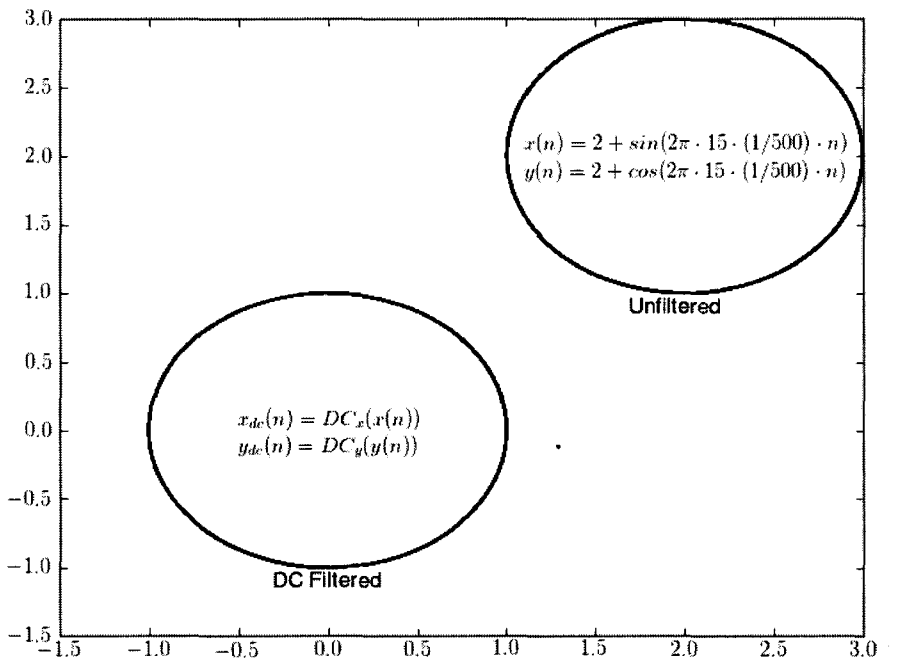

Figure 2.10: Unfiltered and DC Filtered Orbits

shape of the orbit.

Figure 2.10 does not actually tell the entire story of DC filtering. As with most filters, the DC filter has a settling time when first used. In particular, the first 300 results from DC_x and DC_y have been removed, for clarity.

Figure 2.11 illustrates this settling time. It shows the first 300 results of $y_{dc}(n)$ as they relate to the first 300 samples of $y(n)$. By sample 300 the filter has settled and its results can safely be used.

As a comparison to figure 2.10, figure 2.12 nicely shows the plot of the orbits when settling is not taken into account. A system implementing real-time plots of DC filtered orbits must consider this, otherwise users will see the spiralling on startup. Users of the system we developed for the case study were confused by the spiralling when shown it, and expressed preference for a version that takes settling into account, even when that means it takes a short amount of time before values are allowed to be drawn to screen.

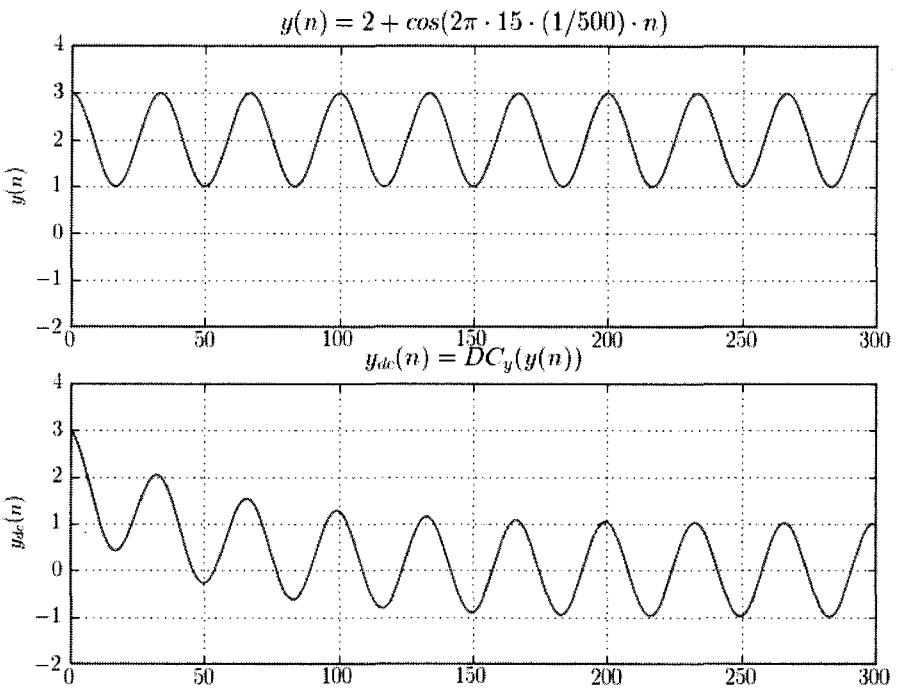

Figure 2.11: First 300 Output Values of DC_y

2.6 Butterworth Filter

The Butterworth filter is often used to show a "smooth" view of the orbit of a rotating machine, implemented as a bandpass filter centred around the main operating frequency of the vibrating screen. In systems where the operating frequency is explicitly known beforehand, the filter can simply be designed around that frequency. When it is possible for the frequency to vary, the frequency has to be determined at runtime using the RPM calculation technique from section 2.7. The system must then be able to dynamically generate a new filter based on this frequency.

The Butterworth filter is an infinite-impulse-response filter chosen for its characteristic of being mathematically maximally flat in the passband region[7]. Centred around the operating frequency of a rotating machine, the passband region can be used to show the fundamental motion.

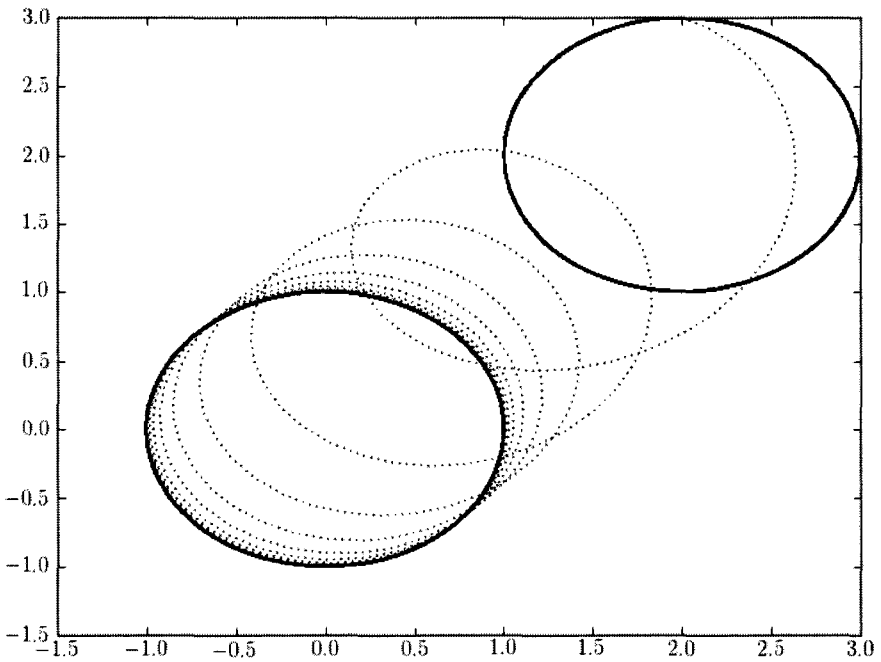

Figure 2.12: DC Orbits Without Taking Settling Into Account

While other filters, such as Chebyshev[8] exhibit faster roll-offs, roll-offs that would require increasing the order of the Butterworth filter to match, the flat pass-band and monotonic frequency response of the Butterworth filter is more important for VA.

In the developed VA system, the acceleration values output from the Butterworth filters are used for various calculations in the software, including maximum G and average G, so a gain of 1 and no ripple in the pass-band is vitally important. Chebyshev filters have the faster roll-off, but at a cost of ripple in the pass-band. Any ripple in the pass-band will skew the acceleration-based calculations and give an incorrect view of the motion and forces created by the vibrating system under analysis.

The magnitude response of a low-pass Butterworth filter is given by [7]

$$|H(\Omega)| = \frac{1}{\sqrt{1+(\frac{\Omega}{\Omega_c})^{2N}}}$$

where Ω_c is the 3-dB or cutoff frequency and N is a positive integer, the order of the filter. The $(\frac{\Omega}{\Omega_c})^{2N}$ in the denominator ensures that $H(0) = 1$, and that all the derivatives $H'(0) = 1, H''(0) = 1, \ldots$. This means the function is as flat as possible at $\Omega = 0$, without being equal to unity everywhere, and that the filter is maximally flat. The magnitude response is also monotonic in both the passband and stopband, as shown in figure 6.10.

The `buttord` function (available in many engineering software packages such as Numpy[9] and MATLAB) can be used to choose the order of a Butterworth filter, based on desired frequency characteristics. A lower order is better for computational resources, but higher orders have narrower transition bands, as show in figure 2.13. The developed system uses $N = 4$, though this is easily modifiable.

As an example of the Butterworth filter, real data from a running vibrating screen was passed through the Butterworth filter, and the results plotted.

Figure 2.14 shows the first 300 unfiltered X and Y axis data points, $x(n)$ and $y(n)$ from the accelerometer. A great deal of high-frequency noise is visible, making the primary motion of the screen difficult to see through visual inspection. Figure 2.15 shows the result of passing these points through Butterworth filters $Butterworth_x$ and $Butterworth_y$, i.e.

$$x_{Butterworth}(n) = Butterworth_x(x(n))$$

$$y_{Butterworth}(n) = Butterworth_y(y(n))$$

The real utility of smoothing the data is when plotting acceleration orbits. These orbits allow a technician to see the general motion of the machine. The unfiltered orbit is shown in figure 2.16, and while the general shape is present, there is too much noise. This plot contains 10000 data points, representing a full recording session.

Figure 2.17 again shows the acceleration orbit, but using the filtered data. The motion of the machine is clearly visible, with all the high frequency noise having been successfully removed by the Butterworth filters.

The settling time of the Butterworth filter must be kept in mind when actually using it. Figures 2.15 and 2.17 do not show all 10000 points of $x_{Butterworth}(n)$ and

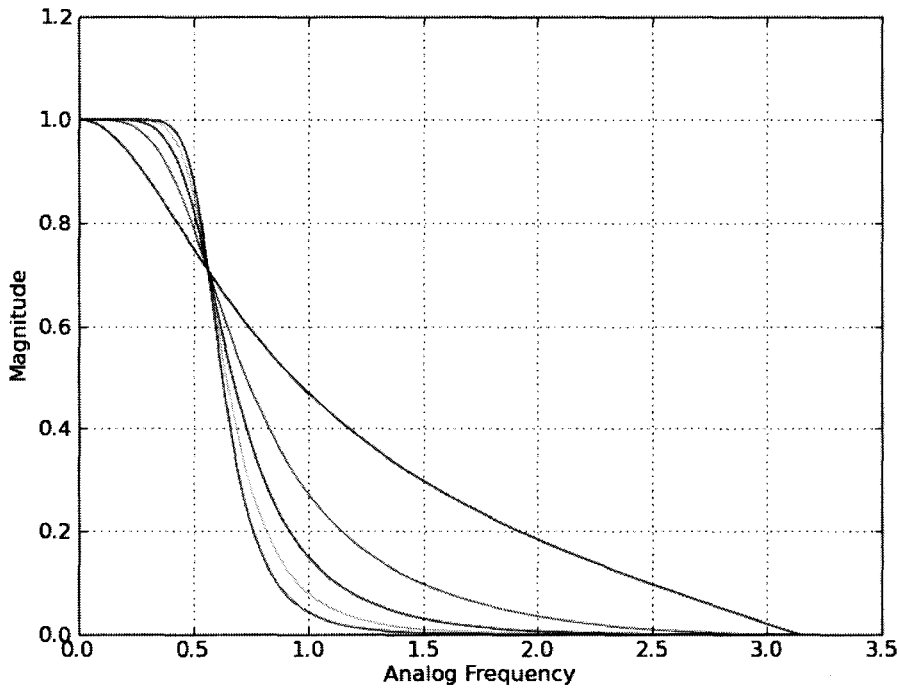

Figure 2.13: Magnitude Response of Butterworth Filters at Orders 1 Through 5

$y_{Butterworth}(n)$. Instead they chop off the first $n = 0 \ldots 500$ points to give the Filter time to settle.

The effects of the settling time are shown in figure 2.18, where there is amplitude oscillation occurring at the beginning of $x_{Butterworth}(n)$. The filter seems to have settled by the time it reaches $n = 200$, the removal of the first 500 points in the software is an arbitrary choice.

2.7 RPM Calculation

In many vibrating systems, the revolutions-per-minute (RPM) that the system is operating at is one of the most important parameters. Most rotating machinery will have a designed RPM, and operating outside of that RPM is usually a sign that the machine is experiencing some kind of fault.

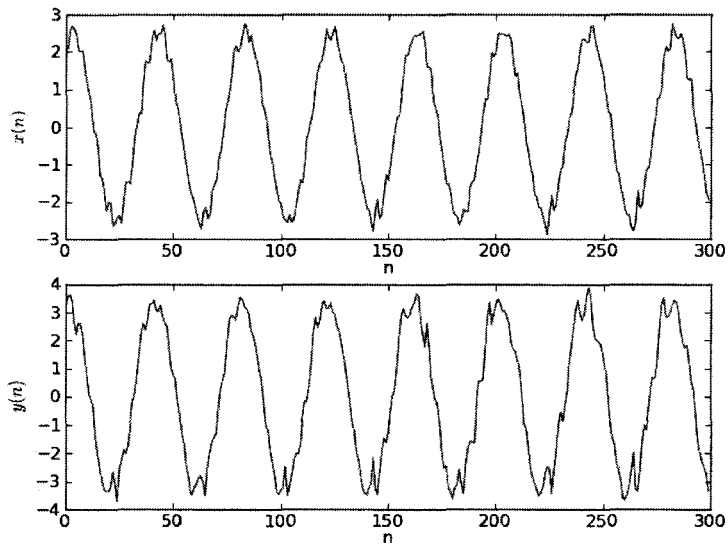

Figure 2.14: Unfiltered Data From X and Y Axes

2.7.1 Frequency Identification

The FFT is the basis for calculating the RPM. The FFT of a set of data is taken, the fundamental frequency component is extracted, giving revolutions-per-second, and then this is multiplied by 60. This basic procedure is illustrated in figure 2.19.

In particular, the fundamental frequency is interpreted as the frequency component with the highest amplitude in the FFT.

More formally, where *argmax* is defined as

$$\underset{x}{\operatorname{argmax}} f(x) = \{x | \forall y : f(y) \leq f(x)\}$$

and the Discrete Fourier Transform (DFT) is defined [10] as

$$X(m) = \sum_{n=0}^{N-1} e^{-jw_0 nm}, 0 \leq m \leq N-1$$

where $w_0 = \frac{2\pi}{N}$.

The frequency component *max_m* with the highest amplitude is

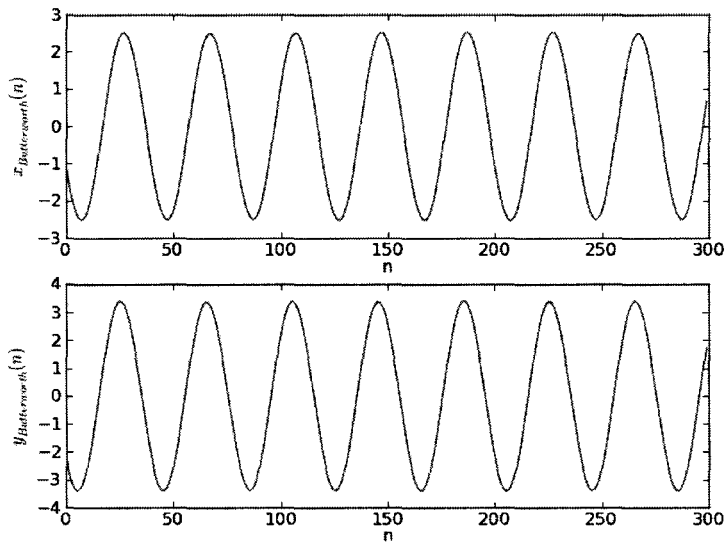

Figure 2.15: Butterworth Filtered Data From X and Y Axes

$$max\_m = \underset{m=[1...(N/2)-1]}{\text{argmax}} |X(m)|$$

While the standard DFT normally operates over frequency bins $m = 0 \ldots N-1$, we need only use $m = 1 \ldots (N/2) - 1$ with *argmax*. $m = 0$ is unneeded because this is simply the DC component of the Fourier Transform, and with the DC filter discussed in section 2.5, there should be no DC present. The upper bound of m can be limited to $(N/2) - 1$ because of DFT symmetry; the DFT outputs for $m = 1 \ldots (N/2) - 1$ are redundant with frequency output values for $m > (N/2)$.

The value for *max_m* is simply a discrete integer index value associated with a particular analytical frequency. The corresponding frequency in Hz can be found with

$$f_{analysis}(m) = \frac{mf_s}{N} \tag{2.1}$$

where f_s is the sampling frequency the data was sampled at and N is the length of the input sequence.

Thus the final value for the RPM is

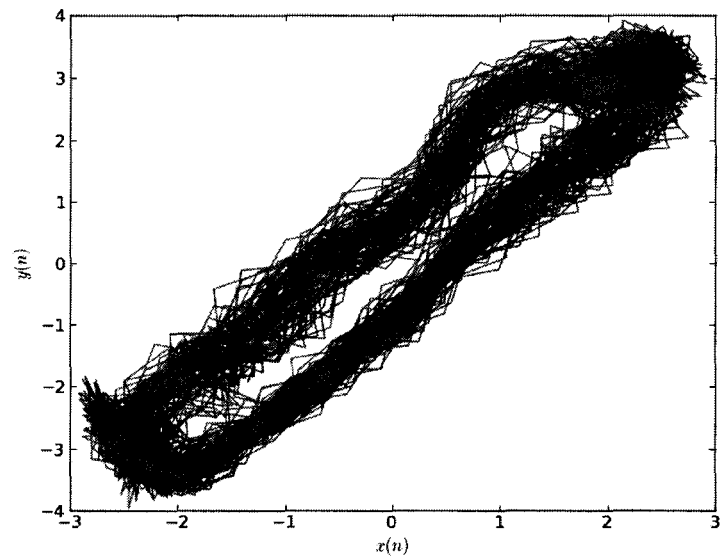

Figure 2.16: Unfiltered Orbit Plot

$$RPM = 60 * f_{analysis}(max\_m)$$

2.7.2 DFT Interpolation

Ideally the value for the RPM from the previous section would be correct as-is. Unfortunately a characteristic of the DFT called "DFT leakage" can result in the calculated RPM being incorrect.

The DFT (and thus FFT) are constrained to operate on N input points sampled from the real-world signal at a rate of f_s, producing an N-point transform whose outputs are associated the individual frequencies of $f_{analysis}(m)$.

Because Equation 2.1 operates on discrete values of m, only N different frequencies are possible, so only N different RPMs can possibly be reported.

The DFT produces exactly correct results only if the frequencies present in the input signal are precisely at analysis frequencies of $f_{analysis}(m)$. If the input signal has any frequency component at some intermediary point between two analysis frequencies, then some energy from that component will appear in *every one* of the N analysis frequencies [10].

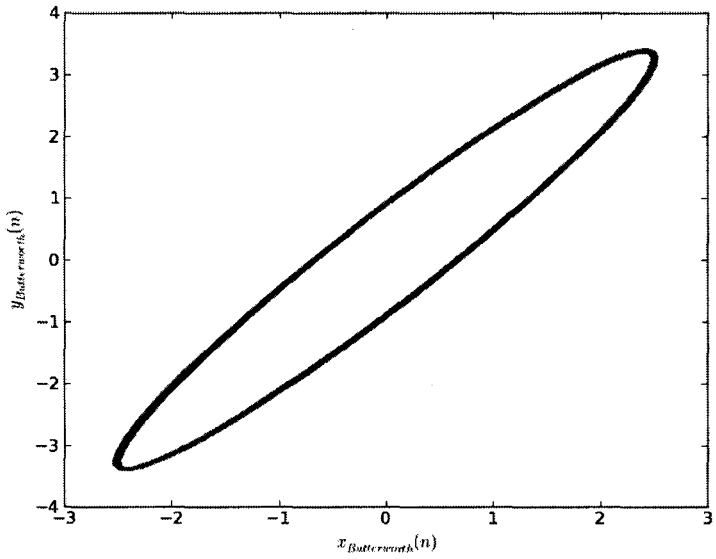

Figure 2.17: Butterworth Filtered Orbit Plot

Figure 2.20 shows two plots. The first is a 64-point input time signal, sampled at 64Hz, generating five complete sinewaves over the 64-points, namely

$$sin(2\pi \cdot 5 \cdot nt_s), \text{where } n = 0 \ldots 63, t_s = 1/64$$

The second plot is the first half of the FFT magnitude of this signal. It shows frequency content at precisely $m = 5$, and no where else.

Figure 2.21 shows two similar plots, but instead of the input sinewave cycling five times, it cycles 5.4 times,

$$sin(2\pi \cdot 5.4 \cdot nt_s), \text{where } n = 0 \ldots 63, t_s = 1/64$$

This small change to the input frequency results in an enormous change to the DFT. Frequency content is shown in all of the frequency bins, despite only a single frequency being present in the input. This is the classic DFT Leakage problem.

Two traditional schemes exist to minimize this issue. The first is to increase the size of the DFT. As the size increases the individual frequency bins become smaller, and the probability of the true frequency fitting exactly into one increases.

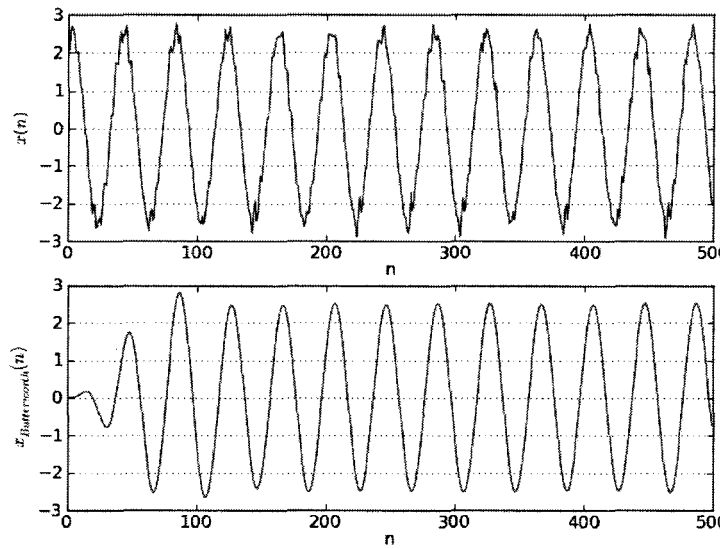

Figure 2.18: Butterworth Filter Settling Time

Of course the size of the FFT must always be finite, so it is impossible to cover every single frequency.

A second scheme is classic DFT windowing. The FFT causes frequency leakage because of its very nature of performing a Fourier Transform over a finite number of samples. Sectioning off a set of samples is equivalent to multiplying an infinite sequence by a rectangular window function. Multiplication in the time domain is convolution in the frequency domain, and the Fourier Transform of a rectangular window is the *sinc* function. The convolution of the true signal with the *sinc* function causes sidelobes, or leakage.

To mitigate this, an alternative window can be used. Figure 2.22 shows the results using two common windows, the Hamming window and Blackman window. Rather than the abrupt change of a rectangular window, these two windows both gradually approach 1.

While the windowing functions drastically reduce the amount of DFT leakage, figure 2.22 shows that it still occurs. Even if it completely eliminated leakage, the result from the DFT would not be correct. The fact remains that if the input signal contains a frequency that lies between two frequency bins, the exact frequency cannot be recovered simply by inspecting the result of the DFT.

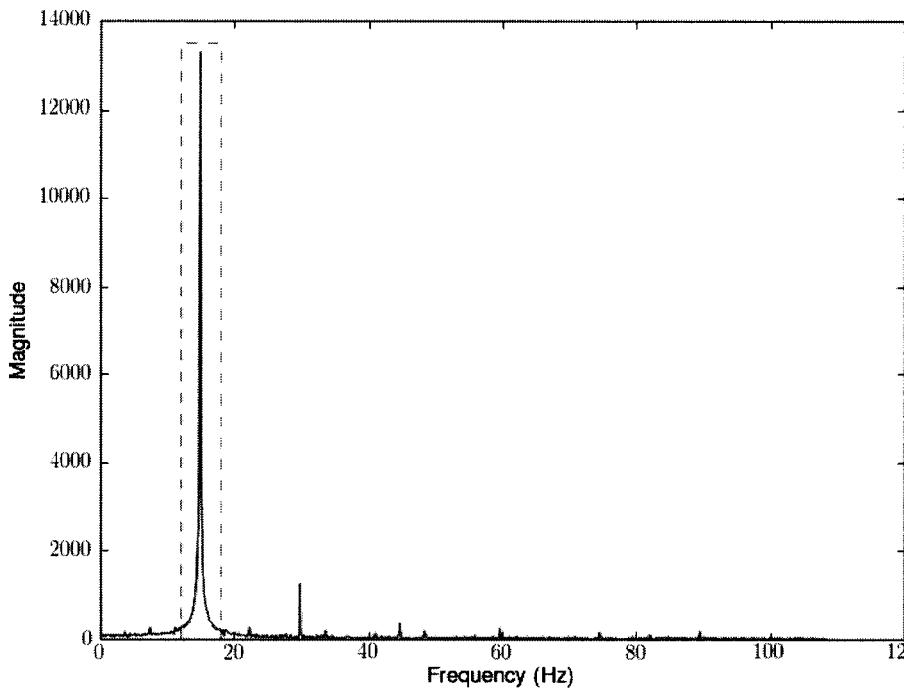

Figure 2.19: Locating Frequency Component with Highest Magnitude

Very high precision was required for the RPM calculation, as the RPM is running at is a key indicator of machine health to technicians. The FFT size could be further increased to reduce the frequency bin sizes, but this comes at fairly severe performance costs. When running with eight sensors, 24 different RPM calculations must constantly be performed (one on each of the three axes for each sensor).

The low-cost solution to this was to perform a polynomial interpolation over the results of the DFT. Centred around the identified frequency (as described in the previous section), a three point interpolation is performed, as illustrated in figure 2.23. This interpolation allows for the identification of frequencies *between* bins, something the DFT cannot do alone.

Using Maple [11], a three-point polynomial interpolation was solved in the general case. This result is

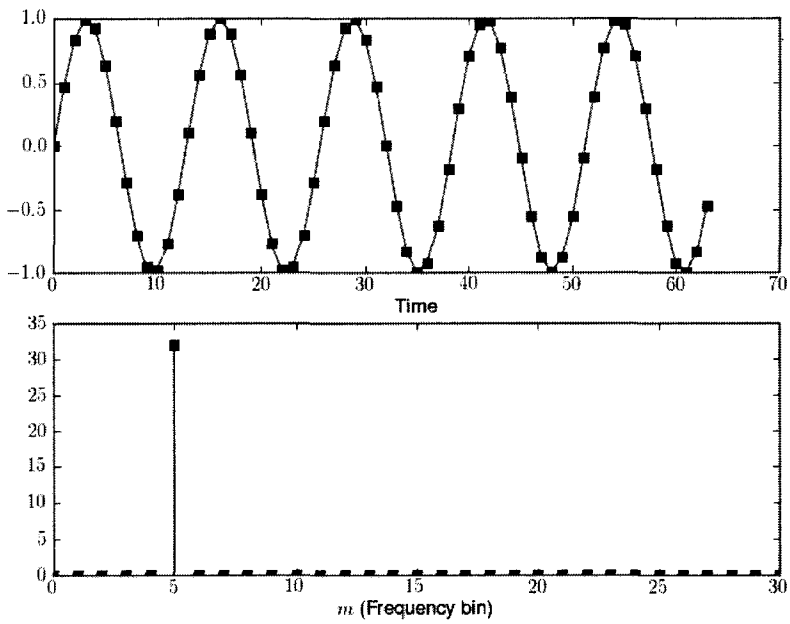

Figure 2.20: 64-point DFT with the Input Signal Precisely at an Analysis Frequency

$$-\frac{1}{2}\cdot\frac{x_1^2y_3-x_1^2y_2-x_2^2y_3+x_2^2y_1-x_3^2y_1+x_3^2y_2}{x_3y_1-x_1y_3-x_2y_1+x_2y_3-x_3y_2+x_1y_2}$$

where

- x_1, x_2, x_3 are the bin numbers, where x_2 is the bin number of the centre frequency (and thus $x_1 = x_2 - 1$ and $x_3 = x_2 + 1$)
- y_1, y_2, y_3 are the corresponding amplitudes

To test this implementation, a simulated input sinewave at 14.4Hz was sampled at 1000Hz. 14.4Hz equates to exactly 864RPM. Performing only an FFT puts the input frequency into the 14.16Hz frequency bin, or 849.61RPM. Putting the FFT result through a polynomial interpolation routine gives an improved 14.369Hz, or 862.183 RPM. Using a Hamming window before the FFT, followed by the polynomial interpolation gives an even better 14.395Hz, or 863.714RPM.

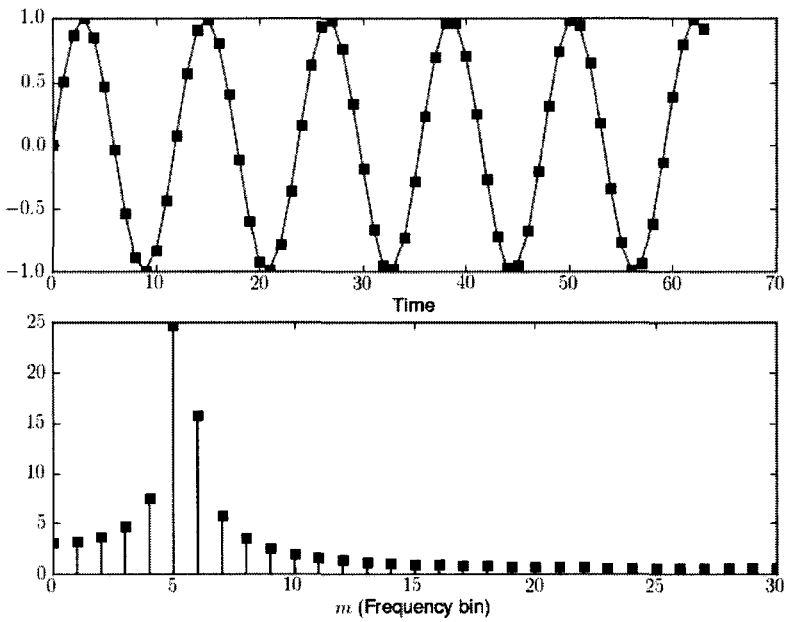

Figure 2.21: 64-point DFT with the Input Signal Between Analysis Frequencies

The polynomial interpolation is a clear improvement over simply using the result from the DFT, and adding a Hamming window improves the situation even more. The question is whether or not the extra cost of a windowing function is worth the improvement, and this is wholly dependent on the circumstances of use.

2.7.3 Potential problem with RPM calculation technique

There is a *possible* drawback with the mechanism used to calculate the RPM, coming from the method used to identify the operating frequency.

The *argmax* function is fairly simple in its operation, and when used with the DFT, it finds the frequency bin with the highest magnitude. Under normal conditions, this frequency will be the operating frequency of the rotating machinery. What if some other frequency component were also present, with an even higher magnitude than that of the operating frequency? *argmax* would instead identify that frequency, and the reported RPM would be completely wrong.

Software implementing this technique could limit the frequency range over

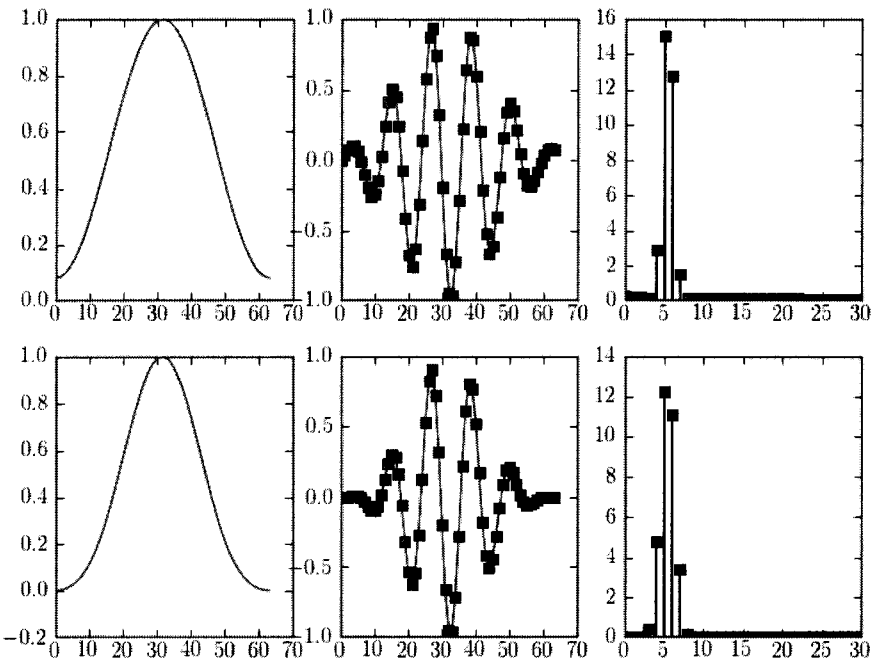

Figure 2.22: DFT Windowing Functions; Hamming and Blackman Windows Applied to the Input

which it calculates *argmax*, but identifying that range is highly dependent on the machine under analysis.

For rotating machinery running at high G-forces, this turns out to not be much of an issue. In a situation like this, if the frequency identified with the highest amplitude is drastically apart from the designed fundamental frequency, then it would often be obvious just from watching the machine operate, without using any VA tools. It would most likely start to shake itself apart.

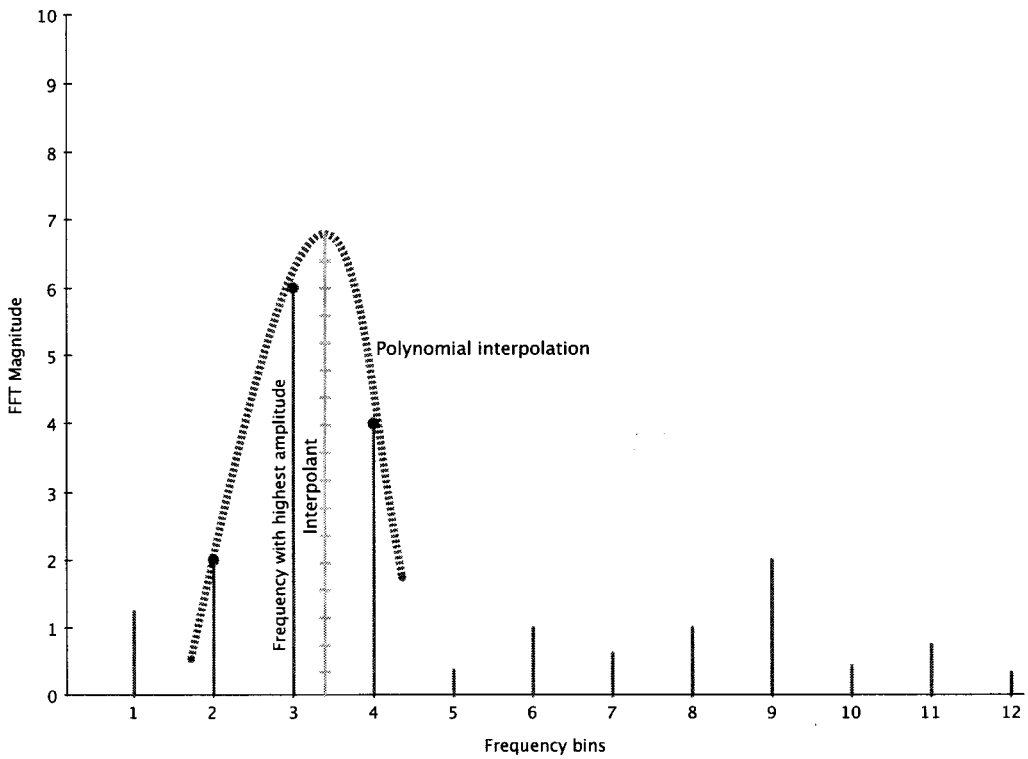

Figure 2.23: Polynomial Interpolation of FFT Magnitudes

Chapter 3

Literature Review

The topics of fault detection and vibration analysis are both areas of heavy research. The ability to monitor structures and machinery – to detect damage or non-optimal operation – is pervasive throughout multiple communities: mechanical, civil, aeronautics, etc. Any mechanical system subject to movement, noise or vibration is a candidate for fault detection and vibration analysis.

Within the field of fault detection, multiple areas of study present themselves and are common to find in the literature. At one end of the spectrum are the highly-mechanical studies of individual components commonly found in these systems. The most common component to study are rolling bearings, vital to almost any rotating machine.

Bearings permit linear motion, or constrained relative rotation, between two parts. Operating conditions can be quite severe, resulting in damage to the bearings. As the bearings are vital to the rotational movement of the machine, a damaged bearing could seriously damage the machine it operates in.

For this reason, early fault detection on bearings is an active topic. Zhang et al. [12] provide a detailed introduction to the problem of bearing faults, and introduce a feature extraction method developed to overcome the limitations of time domain features. The typical time domain features (RMS, peak, kurtosis, crest, etc. [13]) are compared to the frequency domain envelope analysis [14].

Ghafari [15] notes that,

> So far, vibration-based condition monitoring of a rolling element bearing has been mostly studied from a signal processing point of view. Very little attention has been paid to the effect of the fault on the bearings vibration behaviour. Therefore, the first step in successfully

> implementing of bearing health monitoring is to establish the baseline behaviour of a healthy bearing. Furthermore, although a number of rotary machines operate under variable speed and load conditions, very few researchers have proposed robust techniques for the fault diagnosis and prognosis of such systems.

Ghafari proceeds to present the nonlinear dynamics of rolling element bearings, backed up with numerical simulations and experiments. The effects of bearing damage on vibration signatures are then investigated, resulting in the selection of the three fault features least dependent on speed and load conditions. Finally, a neural network decision-making scheme is designed and presented for mapping the conditions into the bearings health.

The use of neural networks, or other automated decision/expert systems is a common theme in the literature.

Despite the large body of work in fault detection, the general consensus remains that VA is still typically manual work performed by experienced technicians [1] [16] [17]. This has led to the use of expert systems to try to automate the VA problem.

More traditional model-based techniques [18][19] require extensive knowledge about the system under study as well as detailed mathematical models for the systems, models that typically do not exist. For very large industrial machinery, a realistic simulation model is very difficult to obtain [20]. Expert systems offer an opportunity to build up a knowledge base for a system, built on direct experience with the system or through another expert.

Yang et al. [17] built an expert system called VIBEX (VIBration EXpert), a decision table based system for diagnosing the cause of abnormal vibrations in rotating machinery. VIBEX works on a set of rules built through IF (symptom) and THEN (cause) statements, with an embedded Bayesian algorithm for obtaining confidence factors.

Another rule based system is presented by Ebersbach and Peng [1]. A knowledge base was constructed to detect the typical faults associated with fixed plant mechanical systems, resulting in a 75 rule expert system with IF/THEN statements similar to VIBEX.

Systems such as VIBEX based on a vibration analyst's basic knowledge can be effective for diagnosis of basic faults, but tend not to perform well against machine-specific anomalies [20]. This is the prime impetus of the machine-learning systems.

Lei et al. [21] present improvements to the artificial neural networks (ANN)

based on an improved distance evaluation technique for optimizing the choice of vibration features to fed into the neural network. The neural networks are trained and used to tune a rule-based fuzzy system to approximate the ways humans process information. The authors' experimental results show promising results in detecting abnormalities within bearings while categorizing and identifying the severity of the faults.

Sun et al. [22] present an ANN customized for pattern recognition of bearing fault signatures, again showing the focus on bearing diagnosis in the literature. Feature extraction happens with the typical time-domain and frequency-domain tools, but also adds in "Segmentation Indices", targeted at analyzing bearing dynamics. This takes advantage of a correlation which exists between the location of a defect within a bearing – inner race, outer race, rolling element, etc. – and the impulse patterns that would be observed through one cycle of the signal. This allows for descriptions of various impulse patterns. The authors detail the effects of the possible defects, and bring the results into the feature extraction, which can be used with an ANN.

A great many more ANN-based systems are present in the literature, with most containing discussions on feature extraction. The ability to identify features in a running system suitable for applying to a diagnosis situation is vital.

The bases of feature extraction tend to revolve around time-domain and frequency-domain analysis [19]. The work by Zhang et al. has already been mentioned, as has the paper by Lei. In addition to the ANN discussed by Lei, the authors also provide a detailed analysis of the features they take advantage of. These include 11 different time-domain parameters and 13 frequency parameters.

Time-domain	Frequency-domain
$T_1 = \frac{\sum_{n=1}^{N} x(n)}{N}$	$F_1 = \frac{\sum_{k=1}^{K} s(k)}{K}$
$T_2 = \sqrt{\frac{\sum_{n=1}^{N} (x(n)-T_1)^2}{N-1}}$	$F_2 = \frac{\sum_{k=1}^{K} (s(k)-F_1)^2}{K-1}$
$T_3 = \left(\frac{\sum_{n=1}^{N} \sqrt{\lvert x(n)\rvert}}{N}\right)^2$	$F_3 = \frac{\sum_{k=1}^{K} (s(k)-F_1)^3}{K(\sqrt{F_2})^3}$
$T_4 = \sqrt{\frac{\sum_{n=1}^{N} (x(n))^2}{N}}$	$F_4 = \frac{\sum_{k=1}^{K} (s(k)-F_1)^4}{KF_2^2}$
$x(n)$ is a signal series for $n = 1 \ldots N$	$s(k)$ is a spectrum for $k = 1 \ldots K$

Table 3.1: Feature Parameters

A small sampling of these features are presented in table 3.1.

A common recurrence with the feature extraction discussions in the literature is that they tend to focus on single measurement points. We propose that the unique feature detection of Chapter 4 and the corresponding vibration localization tool we have developed can be used as an additional feature. This tool generates unique patterns for the spread of a vibration component throughout a machine, which could be used for damage and fault detection. As described in [23],

> Because changes in modal properties or properties derived from these quantities are being used as indicators of damage, the process of vibration-based damage detection eventually reduces to some form of a *pattern recognition problem.*

The patterns resulting from our tool could be used to classify and identify fault situations for that class of machine.

Besides the standard time-domain and frequency-domain feature extraction techniques, a new technique found in the literature is that of the wavelet transform [24] [25] [26] [27]. Not only is this technique being used successfully for fault detection in VA, but also in many other fields [28] [29] [30].

In terms of the frequency-domain, a negative aspect is that one cannot see the time-evolution of frequency components. The short-time Fourier transform is able to account for this, generating spectrograms with time as the independent variable and frequency as the dependent, but the drawback is that the choice of window length affects both time and frequency resolution [31]. However the wavelet transform enables high resolution observation of the evolution in time of the frequency content of a signal.

Wavelets are basis functions derived from one mother wavelet by dilation and translation [32]. They are used to decompose a signal into different frequency components, and study each component with a resolution appropriate to its scale. Low frequencies have more time resolution but less frequency resolution than high frequencies, and vice versa.

This proves very useful for non-stationary signals, where the frequency content might vary over a short period of time.

In addition, wavelets have been successfully used for de-noising vibration signals.

In any measured system, but especially vibrating systems, noise will be present. The ability to attenuate this noise is vital to fault detection.

Mallat [33] was the first to use wavelets for de-noising, while the method proposed by Donoho [34] is considered efficient for Gaussian noise. However these

and many of the other reported methods are not suitable for gearboxes or roller bearings (i.e. typical VA targets) because measured impulses are not smooth. The Morlet wavelet [24] is more suited to this task.

Lin and Qu [24] detail the steps necessary to apply the wavelet transform to de-noising vibration signals with possibly low signal-to-noise rations. They introduce the use of the Morlet wavelet to makeup for deficiencies of the Donoho technique when working with mechanical dynamic signals. This paper does not describe how to setup the thresholds required for the wavelet, a task instead presented by Lin et al. [26]. The authors had positive results for de-noising a mechanical vibration signal with a low signal-to-noise ratio.

Singh et al. [27] apply the wavelet transform to a field of VA not yet discussed, namely the identification of electrical faults in an induction machine. Luo et al. [31] more closely analyze the use of wavelets for monitoring bearings, with good success detecting small defects that otherwise would go unnoticed.

With regards to using wavelets for noise removal, and in fact to all the systems described so far, an assumption has always been made that a single time series signal recording is available. Few systems found in the literature attempt to make use of multiple sensors operating simultaneously on a vibrating system.

The system we propose takes advantage of cross-correlation to act as a de-noising filter, capable of extracting frequency content from a sensor that would otherwise be completely buried by noise. The secondary result of this is that we are capable of building a "vibration flow" diagram for a system, showing how the relative strength of an individual vibration component varies over the surface of a vibrating machine.

When cross-correlation is used for VA, it tends to be in the more traditional form of the cross-correlation function for similarity [35], and for checking time lags [36].

To enable the use of cross-correlation for noise filtering, we had to develop a system capable of running multiple sensors simultaneously. Clayton et al. [36] also developed wireless accelerometer-based sensors, but they are only capable of measuring up to 2G and are limited to two axes. They also present very little information as to the actual design of the sensors and their data acquisition system, information which would be very useful to other researchers building similar systems. This thesis presents detailed design and implementation notes on both the sensors and the DAU.

Vollmer et al. [37] present a "construction kit" for building low-cost VA sensors. These sensors are more capable than ours in terms of maximum G rating and sampling rate, but are not wireless, and once again, do not go into much detail

in terms of design and implementation concerns for building such sensors. The simple act of moving to a wireless system introduces a raft of additional design considerations.

Chapter 4

Unique Feature Detection

A primary objective of VA is the identification of faults or features in a vibrating system. An excellent review of current techniques is available in [38].

Noise of varying degrees is normally present in vibrating systems, sometimes with incredibly high amplitudes, and we wish to provide a means for filtering out this noise to extract useful signal information. Let the true signal of the system be $s(n)$, which is corrupted by additive noise $e(n)$. The observed signal $x(n)$ is calculated as

$$x(n) = s(n) + e(n)$$

From this observed signal we wish to extract $s(n)$.

In a typical filtering situation, a filtered signal $\hat{s}(n)$ is calculated via a filter $g(n)$ through convolution,

$$\hat{s} = g * x$$

for some filter $g(n)$.

In an ideal case, the target properties of $s(n)$ are known beforehand and $g(n)$ can be constructed appropriately to filter out everything except those properties.

Often though these properties are not known beforehand. A typical objective of VA techniques is to explore the frequency characteristics of $x(n)$ (although work does exist on time domain analysis [39]). Depending on the characteristics of the system being analyzed and the noise inherent to that system, interesting frequency content in $s(n)$ might end up being buried by $e(n)$, and will not appear in a simple FFT of the signal.

We propose to use the properties of cross-correlation to aid in the recovery and identification of buried signal components (where $s(n)$ is composed of components of multiple frequencies) in a sensor-network scenario, where multiple measurements of a single system are available. Cross-correlation will be presented as an ideal tool for the removal of noise when Fourier-based frequency analysis is the desired method of signal analysis.

As an application of this theory, a tool has been built for the purpose of frequency-based fault detection. The tool itself is designed for the vibrating screen scenario used throughout the rest of the thesis, but the principles are applicable to any VA situation in which our concept of frequency location detection is applicable.

4.1 Cross-Correlation as an Ideal Filter

Cross-correlation is a measure of the similarity between two signals, with respect to different time lags. This is often used to look for the amount of time it takes for one signal to propagate in another signal, or simply to look at how strongly two signals resemble each other.

For the purposes of using cross-correlation for filtering, some assumptions are necessary.

The first is an assumption on the system under study; noise is additive rather than cumulative.

For the algorithm itself, we make three assumptions:

1. Any frequency that is not visible at more than one measurement point will not be detected by this technique

2. If at one measurement point a sinusoidal signal with amplitude A, frequency f and phase τ_a is present, and at another measurement point a sinusoidal with the same frequency, but with amplitude B and phase τ_b is present, we will say that the same signal is present at both points, regardless of their differences in amplitude and phase

3. We are only attempting to detect periodic features. One-time events measured during a recording period will be ignored

We define a noise filter to be "ideal" when it only removes frequency content that is not visible at two different measurement points, with the condition that

only the frequency itself, and not the amplitude or phase, must be the same at both points.

Thus "noise" is anything that is uncorrelated between two points.

Cross-Correlation as a Noise Filter 4.1.1. *Cross-correlation is an ideal noise filter*

Various properties of cross-correlation must first be illustrated before the idea of using it as a filter becomes apparent.

4.2 Cross-Correlation

Cross-Correlation If two processes are wide sense stationary, the cross-correlation is the expected value of the product of a random variable from one process with a time-shifted, random variable from a different random process [6]

4.2.1 Definition of Cross-Correlation

A nice introduction to cross-correlation and random processes is presented in[40], and will be briefly summarized here.

When dealing with real, discrete signals $x(n)$ and $y(n)$, as we are with the sensor system, the cross-correlation formula is

$$R(x,y)(m) = \sum_{n=-\infty}^{\infty} x(n)y(n+m)$$

where R is the cross-correlation operator and m is the applied lag.

4.3 Cross-Correlation Theorem

The core to the entire filtering operation is the cross-correlation theorem, which itself is dependent on the Fourier transform. As a reminder, the Fourier transform of $x(n)$ is denoted as $\Im\{x(n)\}$ with the result being the frequency domain value $X(k)$. A Fourier transform pair is often written as:

$$x(n) \leftrightarrow X(k)$$

Cross-Correlation Theorem 4.3.1. *The Discrete Fourier Transform of the cross-correlation $R(x,y)$ is equal to the product of the complex conjugate of the Fourier transform of x and the Fourier transform of y, $\overline{X}Y$.*

In other words,

$$R(x,y) \leftrightarrow \overline{X}Y$$

The proof of this requires two other components,

- Convolution theorem
- Conjugation and reversal relationship

First, the convolution theorem,

Convolution Theorem 4.3.1. *The Discrete Fourier Transform of the convolution $x*y$ is equal to the product of the individual Fourier transforms X and Y.*

i.e.

$$x*y \leftrightarrow X \cdot Y$$

where $x*y$ is defined as

$$(x*y)(n) \triangleq \sum_{m=0}^{N-1} x(m)y(n-m)$$

The proof follows from[41], where DFT is the Discrete Fourier Transform operator.

Proof. Let $w_0 = \frac{2\pi}{N}$,

$$\begin{aligned}
DFT(x*y)(k) &\triangleq \sum_{n=0}^{N-1} (x*y)_n e^{-jw_0nk} \\
&= \sum_{n=0}^{N-1}\sum_{m=0}^{N-1} x(m)y(n-m)e^{-jw_0nk} \\
&= \sum_{m=0}^{N-1} x(m) \sum_{n=0}^{N-1} y(n-m)e^{-jw_0nk} \\
&= (\sum_{m=0}^{N-1} x(m)e^{-jw_0mk})Y(k) \text{ (by Shift Theorem)} \\
&= X(k)Y(k)
\end{aligned}$$

□

Definition We define the operator *FLIP* as one that takes a sequence $x[n]$ of length N and returns that sequence in reverse order, i.e.

$$\forall i : (0 \leq i \leq N \Rightarrow FLIP(x)(i) = x(N-i))$$

The *FLIP* operator has a variety of properties relating it to the complex conjugate and Fourier Transforms. One in particular is required in building the cross-correlation theorem:

4.3.1. *For all $x \in \mathbb{C}^N$, the Fourier transform of $FLIP(\bar{x})$ is the complex conjugate of the Fourier transform of x,*

$$FLIP(\bar{x}) \leftrightarrow \overline{X}$$

Proof. Let $m \triangleq N - n$

$$\begin{aligned} DFT(FLIP(\bar{x}))(k) &\triangleq \sum_{n=0}^{N-1} \overline{x(N-n)} e^{-jw_0 nk} \\ &= \sum_{m=N}^{1} \overline{x(m)} e^{-jw_0 (N-m)k} \\ &= \sum_{m=0}^{N-1} \overline{x(m)} e^{jw_0 mk} \\ &= \overline{\sum_{m=0}^{N-1} x(m) e^{-jw_0 mk}} \\ &= \overline{X(k)} \end{aligned}$$

□

Finally, we can prove the Cross-Correlation theorem

Cross-Correlation Theorem 4.3.2.

$$R(x,y) \leftrightarrow \overline{X}Y$$

Proof.

$$\begin{aligned} R(x,y)(n) &\triangleq \sum_{m=0}^{N-1} \overline{x(m)} y(n+m) \\ &= \sum_{m=0}^{N-1} \overline{x(-m)} y(n-m) \; m \leftarrow -m \\ &= (FLIP(\bar{x}) * y)(n) \\ &\leftrightarrow \overline{X}Y \end{aligned}$$

□

This leads directly to an important property of cross-correlation

Definition The **Periodicity Property** states that the cross-correlation of two periodic signals will generate a third signal containing only frequency components that are common to both of the input signals, i.e.

$$\{m : \mathbb{N} | m \leq (N/2) \wedge |DFT(R(x,y))(m)| > 0\} = \{m : \mathbb{N} | m \leq (N/2) \wedge |DFT(x)(m)| > 0\} \bigcap \{m : \mathbb{N} | m \leq (N/2) \wedge |DFT(y)(m)| > 0\}$$

where N is the length of sequences $x(n)$ and $y(n)$.

This follows directly as a consequence of the Cross-Correlation theorem. Assume two signals are present, $x(n)$ and $y(n)$, each a sinusoid with amplitudes A_x and A_y, phases τ_x and τ_y and a common frequency f.

If we take the DFT of x and y (where both x and y have the same number of samples N), we get

$$X(k) = DFT(x(n))$$

and

$$Y(k) = DFT(y(n))$$

In each of these DFTs, the frequency component f will show up in frequency bins m_x and m_y, respectively. More importantly, it will appear in the *same* bin, i.e. $m_x = m_y$. All other bins will contain only 0 (ignoring the effects of DFT leakage).

When $X(k)$ and $Y(k)$ are then multiplied, the resulting signal

$$X(k)Y(k) = C(k)$$

is the Fourier transform of the cross-correlation, and it will only contain content in bin m. An Inverse Discrete Fourier transform (IDFT) is then applied to give the actual result of the cross correlation. Because the inverse was applied to C, which only contained content at the single frequency m, the resulting IDFT will give some signal with amplitude D and phase τ but with frequency f.

This easily extends to signals $x(n)$ and $y(n)$ that contain multiple sinusoidal frequencies $f_1, f_2, \ldots f_M$. As long as both $x(n)$ and $y(n)$ contain the same components, then the resulting cross-correlation will also contain those components.

What about periodic components that are only present in one of the signals being cross-correlated?

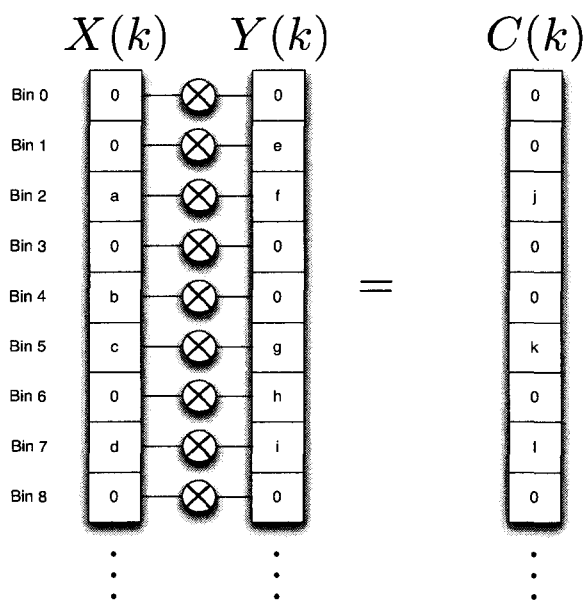

Figure 4.1: FFT of Cross-Correlated Components

Figure 4.1 shows the result. The DFT is taken for x and y, and the results $X(k)$ and $Y(k)$ are multiplied together. Constant values are shown in each of the bins for X and Y. When both have a component in the same bin, then the result $X(k)Y(k)$ also has a result. In every other case, a value is being multiplied by 0. The IDFT of this resulting product will give the cross-correlation, which will contain frequency

content only at the frequencies common to both x and y, as described above for the *Periodicity Property*.

This only holds true if we assume no DFT leakage. In reality, a peak-finding algorithm is required, to look at the product $X(k)Y(k)$ and find actual meaningful content in the frequency bins. The specific algorithm we designed for that is detailed in section 6.4.4.

And an implicit benefit, and one of the goals of this chapter, is that a particular frequency that might be buried in one FFT might be highlighted and extracted when that signal information is cross-correlated with another signal where the frequency is much stronger.

Look again at figure 4.1. Imagine that the value a in Bin 2 of $X(k)$ is so small that it is not visible (i.e. buried) when looking at that FFT alone. Now imagine that the value f in the same bin of $Y(k)$ has a much higher amplitude. Multiplying them together to get value j, and specifically *not* a value of 0 shows that $X(k)$ does in fact contain that frequency. The simulation in section 6.4.3 shows a very nice example of this in action.

In conclusion, we have shown how cross-correlation can be used as an ideal filter for recovering and identifying frequency components in a multi-sensor situation. The remainder of this chapter will explain how to apply this technique to the particular vibrating screen situation of VA.

4.4 Noise versus Features

It was mentioned earlier that if the target properties of a signal are known beforehand, then a filter can easily be constructed which eliminates everything except those properties. For the particular case of vibrating screens, this has often been done with a Butterworth filter centred around the fundamental operating frequency of the screen.

When viewing the sensor data with the filtered-orbit view, as in figures 2.16 and 2.17, the desire is to look at the main motion of the screen. The fundamental operating frequency is the source of this motion, so to centre the Butterworth filter around this frequency causes *all* other content to be removed from the filtered view. Traditionally, this removed content was considered "noise" by technicians. Other frequency content would be looked at in the FFTs (which were performed on unfiltered data), but the issue was that "filtered" data was strictly considered to be data that had all frequencies other than the main operating frequency removed.

The problem with this definition of noise is that any interesting frequencies aside from the fundamental frequency have more or less been ignored.

To make our observed signal $x(n) = s(n) + e(n)$ more explicit than in section 4, we will rewrite it as

$$x(n) = s_m(n) + s_f(n) + e(n)$$

where

- $s_m(n)$ is the fundamental operating frequency
- $s_f(n)$ is the summation of any other interesting frequency content
- $e(n)$ is the noise

So $s(n) = s_m(n) + s_f(n)$.

Ideally, a filter should be available that *only* filters out $e(n)$, and maintains any interesting frequency content $s_f(n)$ as well as $s_m(n)$.

For this technique to filter out the noise $e(n)$, the noise *must* be uncorrelated between different sensor points. The filter operates by simultaneously analyzing two signals and eliminating any uncorrelated content. Any correlated noise might come through. For vibrating screens this will not be an issue, but care must be taken when using this technique in other fields.

If the specific properties of this "interesting" content happened to be known beforehand, then producing a filter like this would be trivial. Unfortunately there is no complete knowledge-base of fault-causing frequencies for every vibrating screen in existence, so instead the filter must be able to dynamically determine what is noise and what is not noise, and act appropriately. This is precisely what our cross-correlation based filter is able to accomplish.

Traditional vibration-analysis on vibrating screens would centre the Butterworth filter around the designed frequency of $s_m(n)$, but this would eliminate all content from $s_f(n)$. So while useful for looking at the fundamental motion of the screen, it does not help with fault analysis.

4.5 Vibration Localization

Vibration localization is the process of determining the physical location that a certain measured frequency component is originating from, or at least determining where the effects of the fault are the strongest. While not applicable to all VA

scenarios, at least with the case of vibrating screens, fault localization can often be performed by finding the area on the screen in which a particular frequency is most prevalent. The way in which that frequency degrades when moving away from that point can be informative as to the nature of the fault itself.

We have developed techniques that can be combined to automatically perform frequency localization for the purpose of detecting periodic unwanted frequency components and presenting the results to a technician in a means more useful that the simple FFT plots typically used. The techniques are also capable of illustrating the presence of vibration in a physical location that would be missed by normal FFT and filtered-orbit data analysis.

4.6 Process Description

As mentioned previously, the cross-correlation of two signals with the same periodic component results in a new third signal with that same periodic component present. This property is especially useful when that periodic signal is so weak in one of the two input signals that it otherwise gets buried by noise when analyzing that signal in isolation.

The two primary means thus far for analyzing signals have been the FFT, and the filtered-orbits.

An ideally performing screen will show the main vibrational driving force, and no other strong frequency components on the FFT. The presence of additional frequency components is typically looked at as problematic, often indicating a fault.

In regards to the filtered orbits, *only* the main motion of the screen is shown. The filters have traditionally been bandpass filtered, centred around the fundamental frequency. While this has always been helpful in removing noise when trying to inspect the main motion of the machine, the problem is that it also removes any actual fault components that might be present.

The problem with using the FFT alone to look at non-fundamental frequencies is that it can be difficult to find common frequencies between the sensors, and a potential fault frequency might be buried by so much noise that it cannot be seen on the FFT.

So we propose using cross-correlation between sensors, and taking the FFT of the cross-correlation. This is a new way of analyzing accelerometer data for vibrating screens, fully separate from the traditional single FFTs and filtered-orbits.

For a particular sensor recording session, let A be the set of individual recordings from the $N = |A|$ sensors. Let G be the group of combinations of A.

We then take the cross-correlation of each of these combinations,

$$CC = \{R(x,y)|(x,y) \in G\}$$

For each of these cross-correlations, we take the DFT,

$$FCC = \{DFT(c)|c \in CC\}$$

Finally we find the set of frequencies such that each frequency in that set appears as a frequency component in more than one of the DFTs,

$$\{m|0 \leq m \leq (N/2) \wedge match(FCC,m) \geq 2\}$$

where *match* is a function that takes in a set like FCC and a frequency bin number m and returns the number of DFTs in that set which have frequency content at that bin.

Note that the description here assumes that all bins other than those with distinct frequency content will contain a value of 0. Because of DFT leakage and other factors, this will typically not be true. After computing the DFTs, an automated peak-finding algorithm is employed, to locate the frequencies which have shown themselves to be present through the cross-correlation. This algorithm is discussed in detail in section 6.4.4.

A tool has been developed which fully automates this process. Given the recorded data files from a VA session, it will calculate the FFT of the cross-correlations between every sensor, automatically discover the relevant resulting frequencies (subject to user-supplied thresholds), and plot the results in a manner usable to technicians. The plot is an undirected graph showing the prevalence of a particular frequency between each possible pair of sensors. An example of one of these plots is shown in figure 4.2

In this figure, a simulation is run with a fault at a particular frequency occurring between locations RFS and RDS on the vibrating screen, with the severity of the fault decreasing as the other sensors move further from this point. The cross-correlation with the highest resulting amplitude for this frequency (in this case the cross-correlation of RFS and RDS) results in an edge between those two nodes with the greatest weight. Any other edges on the plot have weights proportional to the amplitude of that fault in their cross-correlation. As one can plainly see, the weights of the edges decreases as the edges move further away from the vibration location.

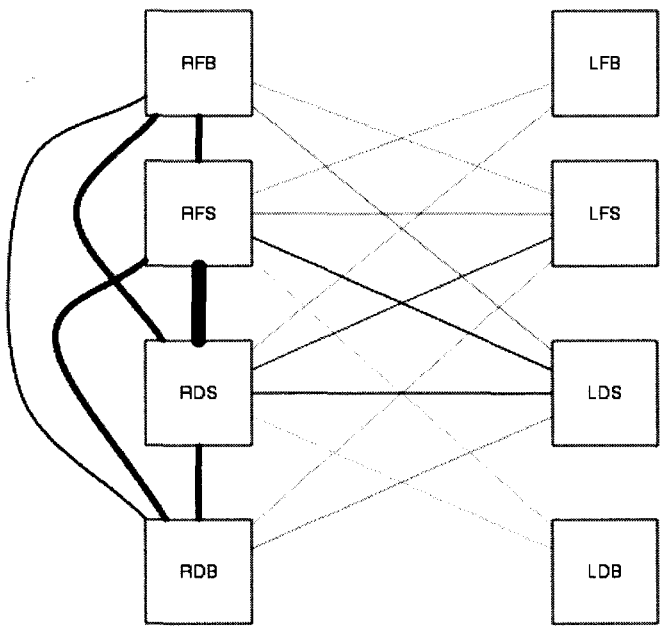

Figure 4.2: Vibration Location Detection; A Fault is simulated in the physical location between sensors RFS and RDS

This tool and the simulation are explained in further detail in section 6.4.

Chapter 5

Design of the Vibration Analysis Sensors

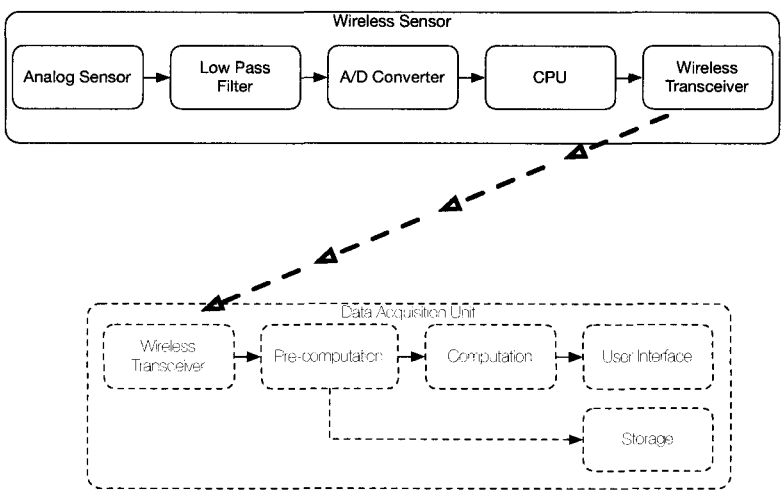

Figure 5.1: Wireless Sensor Components

This chapter describes the design and implementation of wireless sensors used for VA. In terms of the system component diagram, figure 5.1 shows the components that will receive focus here.

This chapter will cover not only the final designs of the components and software, but also the decisions leading up to these designs, problems encountered and solutions to these problems.

The most important contribution of this chapter is the detailed description of the issues surrounding high data transfers over Bluetooth in an embedded system. A full grasp of potential pitfalls requires understanding of the transceiver, handshaking protocols between the transceiver and processor, buffering schemes and atomic memory access on embedded processors.

The hardware section (section 5.3) will describe solely the hardware of the final system, including design decisions and discoveries that lead to each prototype stage.

5.1 Wireless Network Technologies

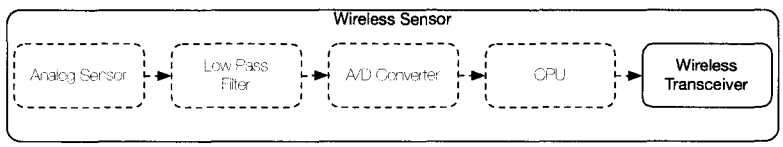

Figure 5.2: Components Affected by Choice of Wireless Technology

An early design decision in a wireless sensor system will be the choice of wireless technology to use. As long as the system is modularized properly, then this choice of technology *should* only affect the overall sensor as shown in figure 5.2.

For wireless communications within a close-range sensor network, there are three primary technologies used today.

- ZigBee [42]
- 802.11 WiFi
- Bluetooth

ZigBee is a wireless technology designed specifically to allow multiple sensors in a small area to communicate with each other. It has a maximum data rate of 20-250 KB/s and very lower power requirements. Unfortunately it is still a somewhat niche product, and desktop/laptop compatible transceivers are still relatively difficult to find.

WiFi is the familiar wireless technology that most computers now come standard with. While it has high theoretical data rates and ranges, its power draw is typically considered too high for standalone wireless sensors.

Bluetooth [43] is a standard wireless networking protocol typically employed in short-range low-power situations. The standard defines different "Classes", discussed in more detail in section 5.3.3, with different range and power ratings.

An interesting technical limitation of Bluetooth is the size of a network. An individual Bluetooth network, called a *piconet*, can only contain eight devices. The system designed here requires eight sensors simultaneously, but also requires the DAU to communicate with them, creating a requirement for nine simultaneous Bluetooth devices. The PDA chosen to act as the DAU comes with a Bluetooth transceiver built-in, so the solution to this problem was to create two separate Bluetooth networks.

The Bluetooth standard does allow for the concept of *scatternets* [44], a mechanism wherein several small Bluetooth networks can be linked together into a larger network topology. This would have allowed for eight sensors and the DAU, without requiring the extra transceiver on the DAU. Unfortunately the availability of Bluetooth devices which support this system is extremely limited.

The creation of a Bluetooth network is currently a fairly manual process, despite some work done towards automatically configured networks [44]. As it stands, a control device will typically initiate a "Discovery Process", transmitting a special message that all nearby Bluetooth devices will recognize and respond to, asking for the devices' names and unique addresses. After determining which devices are the sensors, it can initiate individual connections to each of the sensors. These are akin to TCP-based sockets in a standard networked environment.

The choice of wireless technology will be highly dependent on the particular system under development. For the system developed here, Bluetooth was deemed most acceptable. The drawback of the piconet size could be dealt with, and the power, speed and availability of devices were all enough to win out over the other two contenders.

5.2 Design of the Sensor Software

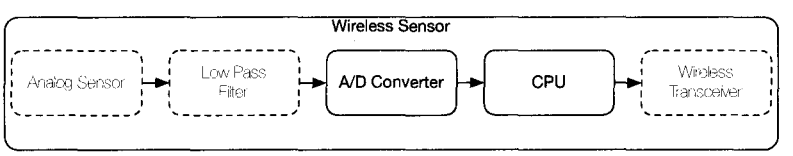

Figure 5.3: Sensor Components Containing Software

This section describes particular design and implementation details of the sensor software. Section 5.2.2 will provide a general introduction to the software components present, and the main operations the software undertakes, to provide the reader with a high-level understanding of the system.

Section 5.2.3 will introduce the issues described at the beginning of the chapter, concerning high data transfers over a Bluetooth sensor network. The particular aspects factoring into the issues will be discussed, but details on each will be saved for their own sections.

At a minimum, the software written on a wireless sensor will affect the CPU component of figure 5.3, but in many cases, including the VA system presented here, the software also controls the A/D conversion process.

Most embedded processors contain internal A/D modules. External A/D converters are usually used for higher precision, but the Microchip PIC18F2523, used in this project, contains a high-precision 12-bit A/D converter, thus making the A/D conversion process part of the software developed for the sensor.

5.2.1 Sensor States

For a wireless sensor tasked with sampling its environment, there are five primary states to be concerned with, as shown in figure 5.4.

In short, the device is turned on, putting the sensor into the `BOOTING`. From here the bootloader waits to see if new software is available for loading, and if not, moves the device to the `ON` state. From here the sensor sits, waiting for a connection from a control device. This connection moves the sensor to the `CONNECTED` state. Often times the control device can configure the sensor from this state, but from a high-level, the control device will tell the sensor to start recording data (putting it in the `SENDING` state) and tell it to stop recording (returning it to `CONNECTED`).

The sensors developed for the VA system include status LEDs, one red and one green, to let the technician know which state the sensor is current in. Their possible states are:

Other than moving from the `BOOTING` stage to the `ON` stage, all visible state transitions are controlled directly by the user through either the physical On/Off button on the sensor, or via the user's DAU.

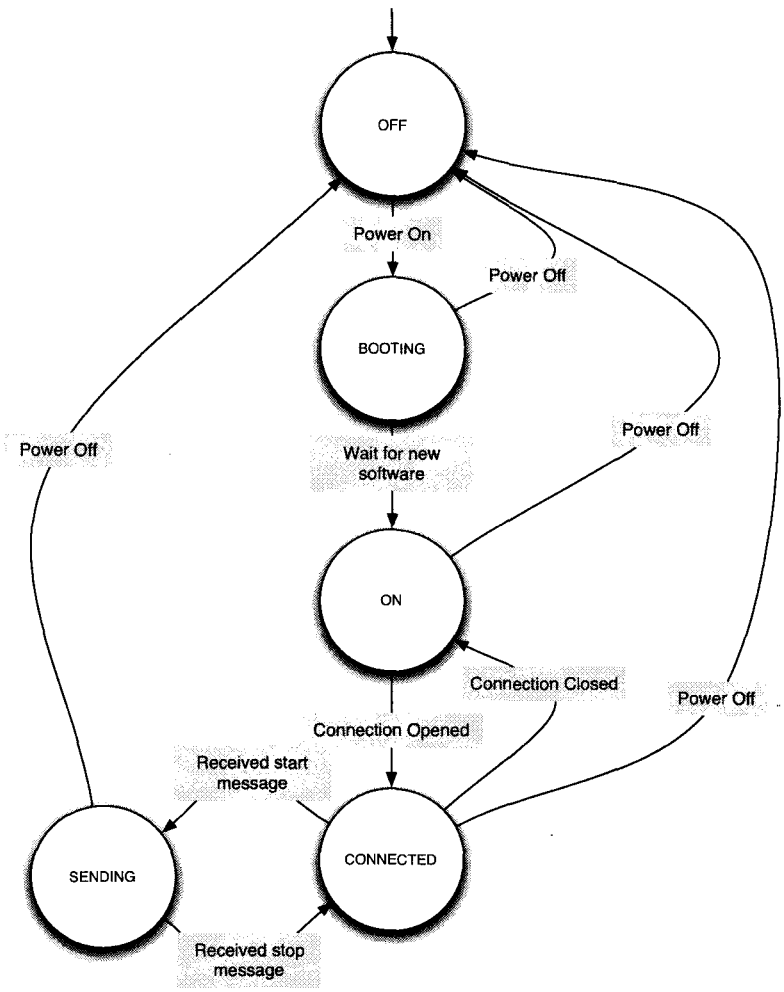

Figure 5.4: Sensor States

5.2.2 Software Operations

In an embedded sensor device, there are six main operations that must take place.

1. Bootloader
2. Initializing processor services
3. Communicating with control device

Red	Green	Meaning	Identifier
Off	Off	Sensor is turned off	*OFF*
Off	Flashing	Sensor is booting, awaiting new software	*BOOTING*
On	Off	Sensor is booted and waiting for a connection	*ON*
On	On	Sensor is connected to the DAY	*CONNECTED*
Off	On	Sensor is sending accelerometer data	*SENDING*

Table 5.1: State Descriptions

4. A/D conversion of analog sensor values (hard real-time)

5. Temporary storage of converted values

6. Formatting and transmitting values over wireless transceiver

Each of these areas will be discussed, and made concrete using specifics from the developed VA system.

Bootloader

The bootloader is the first software module to run with an embedded processor first receives power. It is responsible for loading the main application software and beginning its execution. Often times, the bootloader is also responsible for checking whether new application software is available for the processor, and will load and store this new software.

On the 18F2523, the bootloader is placed in code memory location 0x000 [45], and is immediately executed by the microprocessor. It has two main tasks:

1. Check for new software upload

2. Start the main program if an upload is not initiated

Upon starting, the bootloader first initializes an RS-232 communication channel with the Bluetooth transceiver. It then spends 15 seconds blinking an LED and waiting to see if a particular message arrives over the Bluetooth, specifically one to tell the bootloader that the user wishes to upload new software to the sensor. If this message does arrive, then the bootloader switches to a mode wherein it can receive the new software and save it to the sensor's memory.

If no message arrives over the Bluetooth in 15 seconds, or more specifically the "new software available" message does not arrive, then the bootloader will assume that no new software is available and that it should begin running the main sensor software. The bootloader then instructs the 18F2523 to move its Program Counter to code memory location 0x0800, as specified in the processor's data sheet [45].

Most of the work in developing the bootloader is related to processing and storage of new software from the user. A specific protocol has been defined for this data transfer to ensure the software arrives intact and without corruption.

Initializing processor services

The first steps the main software application must take after being loaded by the bootloader is to initialize system services. These often include, but are not limited to:

- Interrupts
- Timers
- Communication modules
- Status LEDs

Embedded processors tend to require very specific, ordered steps to be taken to bring each of these services up. Interrupt vectors need to be pointed at appropriate interrupt handlers. Timer lengths need to be set, and timer complete operations defined. Communication modules (such as RS-232) get their baudrates set, handshake parameters configured, etc. And usually at the end of the process any status LEDs present on the device are turned on, to notify the user that the software has been initialized successfully.

In the designed system, the process is as follows:

- The real-time sampling timer interrupt is setup to occur every 2ms, and indicates that it is time for a new sample of the accelerometer to be taken. This will be discussed in detail in section 5.2.4.
- The analog-to-digital interrupt is turned on, to signify that a full conversion operation has completed, and the 18F2523 is ready to have the resulting conversion value read out of memory. This will be discussed in section 5.2.5

- The USART module is initialized for RS-232 communication between the processor and the Bluetooth transceiver. The use of RS-232 is expounded upon in section 5.2.8.
- The custom FIFOs which are used for storage of accelerometer data. The FIFO is described in section 5.2.6.
- The final system service to be initialized is the analog-to-digital module. The module is setup, but conversions do not yet begin.

After all of these services are initialized, the software turns on the red LED, indicating that the software has started but does not yet have a Bluetooth connection to the DAU. The full list of states for the LEDs is described in section 5.2.1. At this point the sensor is fully initialized, and enters a loop, waiting for a Bluetooth connection from the DAU.

Interrupts

In general interrupts are used to notify the processor that some pre-defined condition has occurred, without application software having to explicitly check for the condition. One of the most common interrupts, and most important to this system, are timer interrupts.

For almost any wireless sensor system, the sensors themselves will be tasked with measuring their environment at a regular period. Whether or not this period is on the order of milliseconds or hours, timer modules can be used to automatically notify the software that the time has come to sample the environment, rather than the software having to continuously poll a clock to check whether the time has arrived.

The timer on the 18F2523, and most embedded processors, can use interrupts for notification. This requires configuring the interrupt vectors on the processor, pointing individual interrupts at particular "interrupt handlers", custom code for servicing a given interrupt.

The 18F2523 provides two interrupt levels, `low` and `high`. Various services on the 18F2523 can be configured to raise an interrupt during certain events, and these interrupts can be marked as `low` or `high` priority. When an interrupt is raised, the 18F2523 checks if it is a `low` or `high` interrupt, and executes the appropriate interrupt handler. When the handler completes its operation, execution will immediately resume at the code location being executed prior to the generation of the interrupt. If a `high` priority interrupt is already being serviced when a `low` priority

interrupt occurs, then the `low` interrupt will be ignored. If a `low` interrupt is being serviced when a `high` priority interrupt occurs, then execution will immediately jump to the `high` priority interrupt handler. After completion of the `high` priority handler, execution will return to the `low` priority handler.

The system as designed and implemented defines two interrupts:

1. Real-time sampling timer (`low`)
2. Analog-to-digital conversion complete (`high`)

5.2.3 Communication Path Issues

This section presents the general contribution of this chapter, detailing a problem encountered with the communications path, an issue relevant to any wireless sensor network, but particularly to Bluetooth networks.

The communications path used here was fairly simple:

- PIC18F2523 processor transmits bytes to a Bluetooth transceiver via RS-232
- Bluetooth transceiver transmits those bytes to a connected listening device (eg. laptop)

During the development of the sensors, extensive testing was done on the communication path, to ensure that all data transmitted from the processor successfully made its way to the Bluetooth transceiver, through the air, and onto a paired laptop. This was to check consistency of the communication protocol and the communication path.

The test consisted of the processor continuously sending a pre-configured stream of data to the laptop. The laptop could then check that every byte arrived properly, as it knew ahead of time what the data stream should look like.

Early on, very occasional problems were detected, where it seemed that quite randomly a byte would go missing, failing to appear at the end-point of the communication path. At this point in the development, we were fairly certain that it was not related to an actual software bug, but instead some issue with the communication path itself.

The first part of the path investigated was the RS-232 connection between the processor and the Bluetooth transceiver. This was simply two data lines, a "transmit" and a "receive". An oscilloscope was placed on the transmit line, and the

tests were started. When a byte of data was missed on the laptop, the oscilloscope was paused, and the transmit line was inspected. No problems were found here, every byte that was supposed to go from the processor to the transceiver was going over the line correctly.

At that point, the RTS/CTS lines (described in section 5.2.8) were not in use. In short, these lines allow one member of an RS-232 connection to tell the other whether or not it is ready to receive data. The oscilloscope was placed on these lines, and the tests were rerun. This was the first success in tracking down the issue. Whenever a byte of data was missed, the RTS line would be seen going high, indicating that the Bluetooth transceiver was telling the processor to suspend data transfer.

The question then is what was causing the transceiver to raise the RTS line. We soon discovered that it was caused by Bluetooth discovery. In short, if another Bluetooth device nearby happened to initiate a discovery request, the transceiver on the sensor would respond to the request, temporarily halting its transfer of data to the laptop.

What was interesting was that discovery requests did not *always* cause data loss.

Without access to the software for the transceiver we can only postulate as to the reason, but an understanding of typical networking devices lead to the following theory:

> The Bluetooth transceiver implements a local buffer, temporarily storing bytes of data before transmitting them over-the-air.

If true, this completely explains the witnessed behaviour; If the transceiver's buffer happens to be empty, or near-empty, when it begins responding to a discovery request, it can still receive data from the processor. This data will just be held in the buffer until the discovery is complete, and then transmitted. However, if there is already data in the buffer when the discovery response initiates, the incoming data from the processor will quickly fill the buffer, forcing the transceiver to tell the processor (through the RTS line) to stop sending it bytes.

Our solution to this problem was to implement our own local buffer on the processor. All data read from the accelerometer is temporarily placed in our own buffer, and only transmitted to the transceiver when, through the RTS line, the transceiver communicates that it is ready to receive data. Sections 5.2.6 and 5.2.6 present the implementation details of this buffer, which was non-trivial to implement on the 18F2523. The difficulties came from the memory layout of the

processor, as well as the lack of atomic memory operations. The methods for dealing with both of these issues are explained in detail in the mentioned sections.

Very late in the development of the system, another issue related to the communications path was discovered. Surprisingly, we started to see missing bytes of data again, in a similar fashion as before the first problem was fully understood.

What we found was that when the system was running with seven and eight sensors, a situation we could not test until enough prototypes had been manufactured, *too much* data was being sent, and the buffer implemented on the processor was overrunning.

Bluetooth networks operate on a time-slice basis, each device in the network gets a slice of time in which to transmit, before having to stop and pass the turn to another device. What we found was that with a full network of eight devices, the sensors did not have enough time to transmit all the data that was being sent. This would cause the transceivers' buffers to fill, cause them to raise their RTS lines. These lines would stay raised for so long that the buffers on the processors would also fill up, causing identical issues as the discovery problem.

The only solution to this problem was to reduce the amount of data being transferred. This was done both by using more efficiently packed data, as well as reducing the sampling rate. These techniques are described in detail in section 5.2.11.

Neither of the main issues described in this section were anticipated before development had begun, and each cost a great deal of time for diagnosis and development of solutions. Anyone hoping to use Bluetooth for a high data-transfer wireless network must be aware of these issues. The solutions presented here are fairly general, though many of the specifics would require modifying the details to deal with the particulars of selected hardware.

The other idea that should be recognized is the importance of testing with the maximum number of sensors, as early in the development cycle as possible. An extra effort should be made to manufacture additional sensors just for this purpose, as the demands on the network with a full complement of sensors are hard to anticipate without actually testing it.

5.2.4 Timing

The sensor software contains one critical timing component, controlling the sampling frequency of the accelerometer data. These sensors were designed to sample at a rate of 500Hz, therefore a 2ms timer must be present, so that at every 2ms interval a complete sampling of the X, Y and Z axes can be performed.

This 500Hz sampling is a hard real-time constraint, there can be no wavering, as the FFT requires that the data it operates on be sampled at an exact frequency. Shaw [46] defines hard and soft real-time systems:

> A qualitative distinction can be made between hard and soft real-time systems. Hard real-time systems are those that, without exception, must meet their timing constraints–if a constraint is violated, the system fails. At the other extreme are soft real-time systems which can still be considered successful, that is, perform their mission, despite missing some constraints. There is a continuum between the extremes of "hardness" and "softness," and most systems fit somewhere in between.

The 18F2523 provides a means for scheduling hard real-time operations, through its Timer0 module [45]. The Timer0 module is a software configurable 8 or 16-bit timer which can be used to generate an interrupt at a fixed interval. The basic mechanism for interrupt generation is to pre-load an 8 or 16-bit counter with a fixed value. At every clock cycle this counter will be incremented. When the counter eventually overflows, the overflow detection circuitry of the 18F2523 will detect the overflow, notice that it occurred on a special counter associated with Timer0 (specifically `TMR0`), and will raise an interrupt. This interrupt will then be handled by the appropriate interrupt handler.

The interrupt handler for the timer (specifically the `low` priority interrupt handler) has two jobs to complete. It must first reset the `TMR0` counter with the appropriate pre-loaded value, then it must instruct the 18F2523 to begin analog-to-digital conversion. This second operation is handled in its own procedure and is described in section 5.2.5.

It is important to note that that the timer does not stop counting. It is reset to an appropriate value at the beginning of the interrupt handler, and continues counting from there. Thus it is important that the entire analog-to-digital conversion procedure completes in less than 2ms.

The method for determining the value to load into the `TMR0` counter is dependent on the selected clock frequency of the 18F2523 as well as desired resolution. Details on this configuration can be found in Appendix B.1.

To verify that timing configurations in the software were correct, a 100MHz oscilloscope was attached to a sensor and used to measure the time between `TMR0` interrupts. The oscilloscope reported it as exactly 2ms.

Definition of the Scheduling Scheme

Shaw details [46] a variety of scheduling schemes for real-time systems. One of the simplest schemes, and the one that best matches our scheduling implementation is the *foreground-background* model.

In this model, all real-time processes are considered to be periodic. Such a system contains two sets of processes, high-priority foreground processes go into the set FG, while low priority processes go into a background set BG.

The periodic real-time processes are allocated to FG, and are non-preemptible. Processes in BG are preemptible by processes from FG.

Foreground-background systems execute processes from FG periodically, according to a pre-determined schedule. Whenever there happens to be free time available, processes in BG are allowed to execute. If one happens to still be running when the period restarts, then it is preempted.

In our system, the only real-time process is the analog-to-digital conversion, which must take place every 2ms. All other processes, which will be discussed later, are considered non-realtime. They should be executed as quickly as possible, but do not have a hard timing requirement on them. The `TMR0` interrupt ensures the periodicity of the schedule, interrupting whatever process from BG happens to be running, every 2ms.

5.2.5 Analog to Digital Conversion

The 18F2523 contains a 12-bit Analog to Digital conversion module capable of operating on up to 13 different channels, three of which the software makes use of for the accelerometer's three analog output lines (X,Y,Z). While the module can operate on up to 13 channels, it can only perform analog-to-digital conversion on one channel at a time. A special register in the processor is used to select which of the 13 channels should be converted at a given time.

The basic order of operations to perform the three necessary analog-to-digital conversions is:

1. Select the appropriate channel with the `ADCON0` register
2. Start the conversion
3. Put the processor into sleep mode
4. The processor will automatically wake from sleep when the conversion completes

5. Move the resulting value from the ADRESH and ADRESL to the FIFO

6. Select the next channel, and repeat this process

A few of these steps are particularly important to a full understanding of the process.

First, it should be mentioned that the analog-to-digital conversion process is the source of the high priority interrupt. The software configures the 18F2523 such that a high priority interrupt will automatically be raised when the conversion process completes. The handler for this interrupt does not actually perform any useful task. What is its purpose then? The only reason for raising a high priority interrupt upon completion of the conversion is to wake the processor from sleep mode. A configuration setting is available which states that interrupts can be used to wake the processor from sleep.

What then is the reason for putting the processor to sleep? This is done to ensure the highest possible accuracy for the analog-to-digital conversion. The 18F2523 contains two sleep modes, Idle and Sleep. The Sleep mode completely shuts down the CPU and all peripherals, while the Idle mode just shuts down the CPU. Normally these are invoked for power-saving reasons, but in this situation, shutting down the CPU can increase the precision of the analog-to-digital conversion. Shutting down the CPU reduces the possibility of electrical noise from the CPU interfering with the conversion process.

The 18F2523 can be configured so that an interrupt being raised will wake the processor from the Idle and Sleep mode. Waking the processor is the only reason that an interrupt is required on completion of the analog-to-digital conversion process, and explains why the high priority interrupt handler does not perform any actual work. The 18F2523 requires that a handler is present for the interrupt, but the implementation of the handler is equivalent to an empty sub-routine.

When the analog-to-digital conversion process completes, it not only raises an interrupt, it also puts the result into two registers, ADRESH and ADRESL. Two registers are required because the 18F2523 uses a 12-bit analog-to-digital converter, but is only 8-bit processor. Four bits of the 12-bit result go into the bottom half of ADRESH, while the remaining eight bits of the result go into ADRESL. A subsequent analog-to-digital conversion will overwrite these values, so they must be stored somewhere else until the time comes to transmit them. The values are put into the FIFO (described in section 5.2.6), and then the next analog-to-digital conversion process is started.

The time taken to perform all three analog-to-digital conversions and store all the results in the FIFO was measured with an oscilloscope to have a maximum of

80μs. The worst case execution time is easy to calculate, as the datasheet for the processor details the exact execution times of each instruction. Taking branches and conditional statements into account, the worst case time can be shown to be 80μs.

5.2.6 Use of a FIFO

As mentioned in the previous section, a FIFO was used in the software. The primary reason for this is that occasionally the Bluetooth transceiver will be unable to receive data from the processor, and that data must be stored until the transceiver is ready for it. The simplest way to handle this was to always put all data from the accelerometer into a FIFO, and have a separate process routinely check if the transceiver is ready. Whenever it is, have that process pull data from the FIFO and transmit it. Otherwise continue to wait for the transceiver.

The PIC18F2523 architecture and language do not provide a native FIFO, so one had to be implemented in software

The 18F series of PIC processors have their data memory broken into multiple 256-byte banks. It is an 8-bit processor, but uses 12-bit addressing, meaning there is usually no way to access a full 12-bit address in one instruction (though a few special instructs exist which *can* address 12-bits at once). Instead, a bank selector must be used. The basic idea is that the programmer selects which 256-byte bank they wish to use with the Bank Select Register, which accounts for the first four bits of the address, and then uses 8-bit addressing to access the particular one-byte register address they want within the bank.

This makes implementation of a FIFO non-trivial. Bytes have to be stored within the data memory, being careful to check for the right time to change the access bank (every 256 bytes), for both writing to the FIFO and reading from the FIFO. Further checks must take place for wrapping around the end of the valid memory space.

The FIFO must track the current "put" location and the current "get" location, signifying where the next byte should be written to, and what the next byte to be read is, respectively. Bytes will be written to and read from the FIFO at different rates, making it important to track these two locations separately.

The 18F2523 provides three registers, `FSR0`, `FSR1` and `FSR2` (collectively referred to as "`FSRn`" registers) to aid in this process. These are special registers that can store a full 12-bit address, and are matched with a collection of special instructions (e.g. LFSR, CLRF, etc.) that know how to address them. These `FSRn` registers are used to implement indirect addressing, allowing for access to an ad-

dress in memory without specifying a fixed address to the instruction. This is similar to pointers in C-based languages.

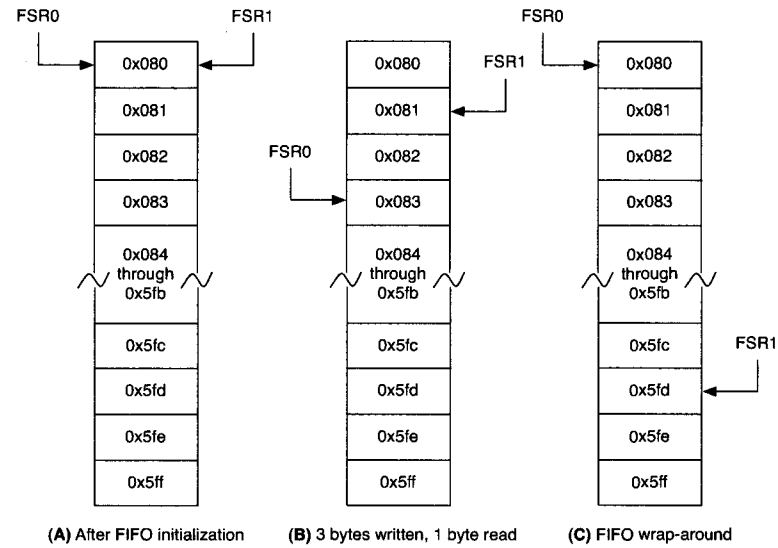

Figure 5.5: FIFO Implementation

`FSR0` and `FSR1` are initially configured to point at the beginning of the FIFO (figure 5.5(A)). Whenever a byte of data needs to be added to the FIFO, it is written to whatever memory location is pointed to by `FSR0`, and then the address in `FSR0` is incremented by one. The `POSTINC0` register shadows the current values of `FSR0`, and will automatically increment itself after an access, simplifying this procedure. A separate counter is also kept counting the number of items currently in the FIFO, which must be incremented here. Finally a routine checks to see if the `FSR0` is currently pointing at the end of the FIFO. If so, it gets reset back to the memory address associated with the beginning of the FIFO, implementing the wrap-around functionality.

The process of pulling a byte from the FIFO is very similar. `FSR1` points to the next byte to be retrieved, which is shadowed by `POSTINC1`, which itself will automatically increment after access. A similar wrap-around check then occurs, and finally the FIFO size counter is decremented.

Figure 5.5(B) shows the state of the pointers after three bytes have been written to the FIFO, and one byte read out.

Figure 5.5(C) shows the wrap-around that occurs. Three more bytes have been written to the FIFO than read, and those bytes caused `FSR0` to require a wrap-

around back to the beginning of the FIFO. `FSR1` will also wrap-around after three more bytes are read from it.

The memory locations shown in figure 5.5 are the actual ones used in the implementation. Though the addressable memory within the PIC18F2523 starts at location 0x000, the first 128 bytes have been reserved for general purpose variables used throughout the software. 0x5FF represents the end of the usable memory.

5.2.7 Atomic Memory Access

Another function that many embedded processors lack is a mechanism for atomic memory access. The FIFO mentioned above is constantly being written to by the analog-to-digital conversion process, and routinely read from for the purpose of transmitting the data within it. Without an atomic lock scheme provided by the processor, a method had to be devised to ensure that no race conditions would be present in FIFO access.

To accomplish this, the deterministic timing nature of the 18F2523 was used. While the timing details are specific to the 18F2523, the techniques used are applicable to any embedded processor. Briefly ignoring the effects of interrupts, each instruction available to the 18F2523 takes a fixed amount of time to execute. This time could be one of a few values, depending on the arguments passed to the instruction and the results of the execution. The datasheet for the processor details all of these timing conditions for each instruction.

Using this information, it is possible to calculate how long a certain sequence of instructions will take, and then verify this calculation using an oscilloscope. The timing requirements for accessing the FIFO could then be exactly calculated. In particular, the key instruction sequences related to reading from the FIFO took the following times:

Instruction Sequence	Time Requirement
Check if FIFO is empty	$1\,\mu s$
Read one byte from FIFO	$3\,\mu s$

Table 5.2: Key Instruction Sequence Timing

Thus the two key components of reading from the FIFO require a total of $4\,\mu s$.

How can this value be used to prevent race conditions? The only way that the full $4\,\mu s$ would not be available is if an interrupt happened to occur during the

time. The only interrupt that could occur during execution would be the Timer0 interrupt, which fires to start an analog-to-digital conversion. As mentioned in section 5.2.4, this interrupt is controlled by a register counter that increments every instruction cycle, `TMR0`. Fortunately, the current value loaded into this counter can be read from software.

The technique is to check the current value of `TMR0`, and if enough time is remaining before the interrupt is going to be raised, pull a byte from the FIFO. If there is not enough time, then the software simply begins to loop back and check again, which will be interrupted by the interrupt. The interrupt is then free to perform the analog-to-digital conversion, and write to the FIFO upon completion. At this point the code to read from the FIFO will resume, find there is enough time until the next interrupt, and do its work.

Using deterministic timing knowledge of the system and the particular conditions of FIFO access within the software, we were able to implement atomic memory access to the FIFO.

5.2.8 RS-232

All data received from the accelerometer must eventually be passed to the control unit via the Bluetooth transceiver. The CPU is responsible for transmitting the data to the transceiver, and the specific method for this is an RS-232 connection between the CPU and the transceiver. The Bluetooth chip selected for this project can only communicate with the CPU via RS-232, but other Bluetooth devices are available that support other protocols such as SPI and I^2C.

Three of the most popular inter-component communication protocols for embedded devices are RS-232, SPI and I^2C Links. The PIC18F2523 supports RS-232 via its Enhanced USART module, and SPI and I^2C via its Master Synchronous Serial Port (MSSP) module. RS-232 is an asynchronous protocol, meaning the clock signal is embedded in the data itself, while SPI and I^2C are synchronous protocols, requiring the presence of a dedicated receive/transmit clock signal. The master device generally outputs this clock signal, which any slave device connected to the master will synchronize on.

RS-232 is a simple point-to-point protocol, allowing one device to connect to another. This provides a great deal of simplicity in setup and configuration, as the two connect via a minimum of only two wires, and need only to agree on a baudrate.

The advantage of both SPI and I^2C over RS-232 is that one can setup a network topology, allowing multiple devices to communicate. With SPI, a single

master device can communicate to multiple slave devices, while I^2C allows for multiple master devices and multiple slaves. SPI allows for multiple slave devices by requiring an individual chip select line to each slave, while I^2C utilizes an addressing scheme, so chip select lines are unnecessary.

A negative aspect of RS-232 is that depending on the device, there is a fairly low maximum baudrate that can be used. With the 18F2523 and selected Bluetooth transceiver, it was found that a baudrate of 115.2kbps was as fast as the chips could be configured before the protocol became unusable. If there was a need to transmit data at a faster rate than that, then either the PIC18F2523 would have had to have been replaced with a more capable processor, or the interface would have had to change from RS-232 to either SPI or I^2C. The Bluetooth transceiver's documentation claims to support baudrates up to 921.6kbps, but the 18F2523 only lists settings up to 115.2kbps.

The baudrate is selected on the 18F2523 via two hardware registers, and the manufacturer provides a formula for calculating the value of these registers based on the desired baudrate [45]:

```
Desired Baud Rate = FOSC/(64 ([SPBRGH:SPBRG] + 1))
```

where `FOSC` is the frequency of the oscillator used with the processor, and `SPBRGH` and `SPBRG` are the baud select registers. This formula then lets the user configure the PIC18F2523 for baudrates higher than 115.2kbps, but in testing it was found that any baudrate higher than that would cause incorrect data transmissions.

As the device only required digital communication between the CPU and the Bluetooth transceiver, RS-232 was chosen. There are a plethora of RS-232 based Bluetooth transceivers available, so the number of devices became the main criteria. Had more components been added that required communication with the CPU, then SPI or I^2C would have been required.

It should be noted that the digital accelerometers are available, those that communicate via SPI and I^2C instead of analog lines. Had one of these been used, then it might have made sense to use SPI or I^2C to communicate both with the digital accelerometer and a Bluetooth transceiver, but nothing would have prevented both from being used at the same time.

An optional aspect of RS-232 are the CTS/RTS lines. These allow a level of "handshaking" between two devices, so one device can tell the other whether or not it is ready to receive data. In particular, one end of the RS-232 connection can raise its RTS output to inform to the other end that it is not ready to receive data, while lowering the line indicates that it is ready to receive data.

A need for this was found in our setup due to the nature of Bluetooth. While in general the Bluetooth chip had no trouble receiving all the data the CPU attempted to send it, it was discovered that in two particular situations the Bluetooth chip needed the CPU to temporarily halt data transfer.

The first of these situations occurs if a Bluetooth discovery is taking place. Bluetooth allows for a general discovery procedure, so one Bluetooth device can gather information on other devices in its vicinity. What was found was that if some Bluetooth device initiated a Bluetooth discovery in range of our Bluetooth transceiver, the transceiver would be forced to stop transmitting data it was receiving from the CPU, and instead respond to the discovery request. This was causing data to "get lost". The CPU would send it to the Bluetooth transceiver, but the transceiver was simply ignoring it so it could respond to the discovery. Fortunately it was found that when the transceiver begins responding to a discovery request it will raise its RTS line, indicating that no data should be sent to it. Simply monitoring this line on the CPU and temporarily halting transmission while it was high was enough to solve this problem. It did require the implementation of a buffer system (the FIFO) on the processor, as a means to temporarily store data from the accelerometer whenever the transceiver sets the RTS line high. After the line went low again the CPU would send all data in the buffer.

The second situation is if the Bluetooth chip is not currently in its transmit time slice, and data must be sent. The Bluetooth chip only has a small buffer, and if it fills up because the chip does not have a time slice and cannot transmit data, then it will raise its RTS line, telling the processor to temporarily stop sending data. This situation ended up causing a problem in the prototyping stages, and is described more fully in section 5.2.11.

5.2.9 Control Communication Protocol

Figure 5.4 represents the main operating states of the sensor, but for clarity, only a subset of states were actually shown in the figure.

The `CONNECTED` state in the figure shows three outward transitions. "Power Off" represents either the power being switched off on the sensor or the batteries dying. "Connection Closed" indicates that the Bluetooth connection was lost, either because it was properly closed, or because it was lost (e.g. other side of connection dying, other side going out of range, etc.). Finally, the diagram shows that if a start message is received over the Bluetooth connection, then the sensor will enter the `SENDING` state.

What is not shown here is the set of other valid characters that the sensor can

receive which will cause it to perform some other action. The total set of control characters are summarized in table 5.3.

Character	Description
{	Start sending data
}	Stop sending data
<	Get battery level
101010XX	G-select byte mask
?	Read EEPROM
#	Write EEPROM

Table 5.3: Valid Control Characters

Full descriptions of the operations performed upon reception of these characters are given in Appendix A.1.

5.2.10 Data Transmit Procedure

After a sensor receives the { character, it moves to the *SENDING* mode and begins transmitting data. This simple operation actually requires quite a few distinct steps:

1. Clear FIFO
2. Clear packet count
3. Adjust LEDs
4. Start Timer0
5. Check if data is in the FIFO
6. Transmit the data

The first step is required to make sure that no previous accelerometer data is transmitted in this particular instance of moving to the *SENDING* state. Clearing the packet count will be described below, as it requires knowledge of the particular packet format. The LEDs are adjusted accordingly as described in table 5.1 and Timer0 is started. This latter point is important to note: If the sensor is not in

the *SENDING* state, then Timer0 is not running, and no accelerometer values are being read. Only when transitioned to the *SENDING* state does Timer0 run, and it is turned off immediately upon exiting the *SENDING* state.

The fourth step makes explicit what is described in section 5.2.4: There is typically more time available in each of the $2ms$ periods than is necessary to send all the data in the FIFO. This "spare time" will often be needed though, and this will be described below.

Figure 5.6 illustrates the exact manner in which packets of data are transmitted from the sensor to the DAU.

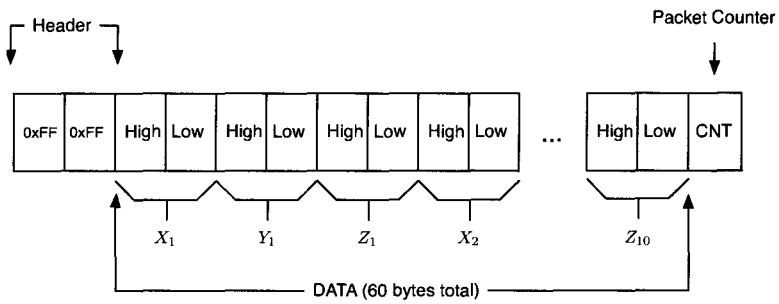

Figure 5.6: Data Transfer Protocol Packet; Each box represents one byte

The first part of each packet is a two byte header, $0xFFFF$. This is used to signal the start of the packet because there is no other place in the packet that it is possible to have the byte $0xFF$ sent twice in a row. Two bytes of data are sent per axis, per sample, but these two bytes of data (16 bits total) are used to store just 12 bits (from the analog-to-digital converter). So there are four unused bits in each two byte sample, and these are explicitly zeroed out. By zeroing them out, the software has ensured that no two subsequent bytes in the `DATA` section of the packet will have the value $0xFFFF$.

The `DATA` section represents the results from 10 sequential sample periods, where the result from each sample period consists of the results for each of the X, Y and Z axes. This can be interpreted as transmitting ten three-tuples in a row

$$[(X,Y,Z)_{t+1},(X,Y,Z)_{t+2},\cdots,(X,Y,Z)_{t+10}]$$

where t is the time point of the last three-tuple in the previous transmitted packet of ten samples.

The final component of each packet is a one-byte counter. When the sensor moves to the SENDING state, this counter is cleared, as mentioned above. It is then incremented once per packet, and placed at the tail of the packet. This is used as a very simple CRC mechanism, so the DAU can check that no packets have been dropped, and that

$$current\_counter\_value = previous\_counter\_value + 1$$

This counter is allowed to roll over past $0xFF$ to 0 every 256 packets, and continue on from there.

As measured (using an oscilloscope) for a baudrate of 115.2kbps, it takes 80μs to read one byte of data from the FIFO and transmit it through the USART, 3μs of which is spent on the read portion, and 77μs on the transmit. A single packet contains $2+60+1=63$ bytes of data, 60 of which include a FIFO read (the 2-byte header and the tail do not need any data from the FIFO). The total time required to transmit one full packet of data is then

$$\begin{aligned} 2\times 73\mu\text{s} + 60\times 80\mu\text{s} + 1\times 73\mu\text{s} &= 5019\mu\text{s} \\ &= 5.019\text{ms} \end{aligned}$$

Typically though during an individual sampling period, only 6 bytes will be placed in the FIFO (two bytes each for X, Y and Z). So during a single period the time taken to transmit data will be

$$\begin{aligned} 2\times 80\mu\text{s} &= 160\mu\text{s} \\ &= 0.160\text{ms} \end{aligned}$$

Thus a full packet will be transmitted over the course of 10 sample periods, where the first sample period transmits the header and the data, the second through ninth periods transmit just data, and the tenth period transmits data and the packet counter tail.

In section 5.2.5 the entire analog-to-digital conversion process was shown to take 80μs. So a typical 2ms period (specifically the second through ninth periods) will require $80\mu\text{s} + 160\mu\text{s} = 240\mu\text{s}$.

This leaves 1760μs of time normally unused in a typical sample period. The software will spend this entire time checking to see if more data is available in the FIFO. There *is* a situation in which more data will be present.

There are two scenarios where this extra time is required. As mentioned in section 5.2.6, if a Bluetooth discovery process is taking place, or the transceiver does not currently have an available transmit time slot in the Bluetooth network, then the Bluetooth transceiver will temporarily be unable to transmit data. The 2ms sampling periods will continue, and the analog-to-digital process will continue to sample the accelerometer and place data in the FIFO, and the FIFO will slowly start to fill up. After the discovery completes and the transceiver can begin to transmit again, the software will start to send all the bytes that are waiting in the FIFO. With 1760μs available, this allows for an extra 22-bytes to be sent, which translates to just under 4 sample periods (8ms) worth of data. This free time in each subsequent sampling period will continue sending bytes sitting in the FIFO until it is one again empty.

5.2.11 Sampling Rate Selection

During the initial design phases, a desired sampling rate of 1000Hz was selected, such that the new system would match the old one. The accelerometers were selected with this requirement in mind and the sensor software was written to implement the 1000Hz rate.

During the design and early prototyping phases there were never more than three working sensors that could be used for testing. This presented a constant worry, an uncertainty as to whether something might fail when the full complement of eight sensors were used.

A first manufactured prototype run was eventually completed, providing eight sensors to test with. Unfortunately this immediately showed issues. With eight sensors running simultaneously, each individual sensor did not seem to be receiving a long enough time slice to transmit all of its data. As mentioned in section 5.2.8, if the Bluetooth chip's own buffers fill up because it does not have enough time to transmit, then it will raise its RTS line high, instructing the sensor's CPU to stop sending it data. If this line stays high for too long, then the sensor's own FIFO will overflow. causing incorrect data to be transmitted.

The FIFO itself does not maintain any information as to what an individual byte of data within it represents, it is simply a collection of bytes written to by the analog-to-digital conversion process. The software knows the order that bytes are supposed to be written into it, and this information is used to interpret the data in

the DAU. If the FIFO begins to overflow then the bytes within the FIFO no longer represent the known order of bytes that result from data collection. The software that reads from the FIFO and transmits the bytes has no idea that anything has gone wrong, and simply continues to transmit. This results in incorrect packets being sent to the DAU.

A few steps were taken to solve this problem.

The first was to change the packet format. In the current implementation, a single packet contains ten sets of data samples, as shown in figure 5.6. The original versions put just one set of data samples into a single packet. With the current scheme, sending 10 sets of samples (one full packet) requires transmitting 63 bytes of data. With the original scheme each set of samples requires 9 bytes total (two-byte header, one byte counter and six bytes of data), so ten sets would require 90 bytes to be transferred. This simple change reduced the bandwidth requirements by a third.

The second solution reduce the sampling rate from 1000Hz to 500Hz. This halved the amount of data that each sensor had to transmit, so the time slices were once again long enough for all the data to be transmitted within. While a higher sampling rate is of course desirable, a rate of 500Hz will work fine in a system with frequency content below 250Hz. 250Hz is still well above the range of a typical vibrating screen, so this trade-off was deemed acceptable.

With these two changes implemented, the data rate was low enough such that eight sensors could be run simultaneously with no problems.

5.3 Design of the Sensor Hardware

5.3.1 Introduction

The final sensors of the system are sealed, dust-proof wireless units capable of transmitting acceleration data to a central control unit via a Bluetooth transmitter. Each sensor has a PIC18F2523 [45] microprocessor which is responsible for performing analog-to-digital conversion on data from a Freescale Semiconductor MMA7261QT ±2.5-10G accelerometer at a rate of 500Hz, organizing the accelerometer readings and transmitting them through the Bluetooth transmitter to the control unit, using the designed protocol (section 5.2.9).

On the exterior of each sensor unit are three high-strength magnets on the backside for the purpose of mechanical mounting to high G-force vibrating machinery. Each unit has a mechanical toggle switch for turning the unit on and off,

as well as two LEDs (one green and one red) for communicating to the user which of the four possible states the sensor is currently in. These states are described in section 5.2).

5.3.2 Sensor Requirements

At the outset of the project, a few main criteria were decided upon for sensor portion of the system. These were:

1. 500-1000Hz sampling rate
2. Capable of running at up to 10G in three axes
3. Dust-proof enclosure
4. High-powered magnetic mounting system
5. Low manufacturing cost
6. Up to eight units running simultaneously

The 500-1000Hz sampling rate was based on understanding of the machines that will typically be analyzed for this particular case study. As described in section 2.1, the operating frequency of most of the vibrating screens is typically in the 14Hz range. Bearing faults can cause frequency content of up to 110Hz, and typically frequency content above 150Hz has been deemed "noise". A 1000Hz sampling rate would allow the system to recognize frequency content up to 500Hz[8], but for reasons described in section 5.2.11 a sampling rate of 500Hz was the eventual rate, allowing the recognition of content up to 250Hz.

Point 2 states that the sensor should be able to read acceleration values up to 10G. While vibrating screens do not typically run past 6.7G, a screen that is malfunctioning could run at higher G-forces. 10G was thus considered to be a "safe" maximum value for accelerometers. The chosen accelerometer can run at up to 10G, and can operate at lower maximum-G settings, to allow for higher accuracy (see sections 5.3.3 and 6.2).

The dust-proof enclosure is required due to the environments in which the VA system will typically be employed, namely active mine sites. A unit that fails due to dust would essentially be useless.

The decision to use high-powered magnets to mount the sensor to the system is based on three facts:

1. High G-forces
2. Mud and dirt coated machinery
3. Hundreds of pre-existing deployed screens

Due to the high G-forces that the vibrating screens operate under it is important that the magnets be strong enough to hold the sensor to the machine without moving. Any movement or vibration of the sensor itself would cause incorrect acceleration readings. Because of the environments that the screens operate in, they are usually covered with dust, dirt and mud, and while the technicians do their best to clean mounting points before attaching a sensor, only so much can be done. Very strong magnets allow the sensor to stay mounted to the machine even in the presence of dirt.

Ideally each vibrating screen would have built-in mounting points for the sensors that provided some kind of mechanical locking system. Unfortunately this is impossible. Hundreds of vibrating screens are already deployed around the world, and the VA system must be useful on all of these machines.

Low manufacturing cost was a key idea that had to be kept in mind at all times of the hardware design process. A system with a sensor network is much less likely to be used or expanded upon if the cost to manufacture it is too high.

Finally the system must be able to operate with eight sensors running at the same time. The reasons for this were described in section 2.1, and this requirement affected not only the wireless transceiver selection but also had consequences on the software design, as described in sections 5.2 and 6.2.

5.3.3 Selected Hardware Components

While the sensors contain dozens of discrete electrical components, the three most important are the accelerometer, the microprocessor and the wireless transceiver. The rest of the components are typical of any electronics project (i.e. resistors, capacitors, LEDs), and were not important relative to the main requirements of the project.

Accelerometer

One of the first decisions made was which accelerometer to use. A variety of accelerometers are commercially available, in one, two and three-axis configura-

tions. An early requirement was that the accelerometer be capable of detecting up to 10G of acceleration.

The chosen sensor is the Freescale MMA7261QT, a Microelectromechanical system (MEMS) component. It is a 3.3V device that can run in maximum-detection modes of ±2.5G, ±3.3G, ±6.7G and ±10G. It can simultaneously read accelerations in the X, Y and Z axes. The sensor operates by outputting three separate voltages, one per axis. The voltage can then be analog-to-digital converted into a digital value, and interpreted based on the selected G-mode. At the time of hardware design, this accelerometer was the only commercially available, low-cost three-axis accelerometer that could measure up to 10G.

In the 2.5g mode, the sensor outputs at approximately 480mV/G, and at 10G mode 120mV/G, so a lower G-mode gives a higher resolution. For this reason, it is important that the software allows the user to select the G-mode, so the most appropriate one can be used for the given application.

The timing of this project was actually advantageous in terms of accelerometer selection. Historically accelerometers have often been built as piezo-electric systems, but recent advances in MEMS technology have driven prices and sizes of MEMS-based accelerometers down to levels such that they can be used in low-cost components [47].

Any similar project would need to closely look at its accelerometer requirements. A huge variety of accelerometers are commercially available, with different communications interfaces, maximum G ratings, numbers of sensitivity modes, etc. For the case study it was desirable to have a multi-mode sensor at 10G, but other projects might differ.

Microprocessor

From the outset, it was decided that using a Microchip PIC microprocessor would be used, due to existing familiarity with the architecture and development tools.

While Microchip manufactures a wide variety of PIC micro-controllers, the 18F2523 was chosen mainly for its 12-bit analog-to-digital converter. The accelerometer outputs its measurements as voltages, ranging from 0-3.3V. A 12-bit analog-to-digital controller can take a 3.3V analog voltage and represent it as a value from 0...4095, or 4096 total values (2^{12}).

Dividing out, we get a resolution of

$$3.3\,\mathrm{V}/4096 = 0.806\,\mathrm{mV}$$

In total, the 18F2523 allows ten of its pins to be dedicated to analog-to-digital conversion. The accelerometer uses three of these, a temperature sensor uses one, and a method of measuring the remaining battery uses another, for a total of five.

The 18F2523 also provides a built-in USART module for RS-232 communications (section 5.2.8) with the wireless module, and 1536 bytes of data memory, and comes in at an extremely low cost. The initial designs of the sensor actually used a chip from the same family, the 18F2423, which is almost identical but with half the data memory. This eventually had to be changed to the 18F2523 to provide more memory for a FIFO, as described in section 5.2.

Bluetooth transceiver

An important issue that should be noted is the "Class" classification used for Bluetooth devices. The Bluetooth specification [43] lists three separate power classes for devices, Class 1, Class 2 and Class 3. The power and range specifications for these are as follows:

Class	Maximum Allowed Power		Range (metres)
	mW	dBm	
Class 1	100	20	100m
Class 2	2.5	4	10m
Class 3	1	0	1m

Table 5.4: Bluetooth Output Power Classes

For the purposes of the VA system, the longest range possible was required. The vibrating screens are often suspended off the floor and difficult to reach. It is easier for the technician to perform VA if they are not strictly required to be proximate to the screen.

With this in mind, the first prototypes of the sensor hardware were built with Class 1 Bluetooth transceivers. It was not discovered until after the prototypes were completed that the range permitted by the chosen device was nowhere near 100m, especially when the signal must pass through the vibrating screen itself.

The problem is that to be considered a Class 1 device, the Bluetooth transceiver need only to have power output higher than 4dBm. A manufacturer can give a device with a rated power output of 4.1dBm a Class 1 designation. This was essentially the case with the first selected Bluetooth used in the early prototypes. More specifically they were at 7dBm, but this was not immediately obvious. A

misunderstanding of the class designations caused us to believe that Class 1 meant that the device ran at 20dBm. The Bluetooth device used in the final production units is a 14dBm device. This is the most powerful device that could be located and sourced, and proved to be powerful enough for the Bluetooth signal to pass through even the largest vibrating screens.

For a Class 1 device to be effective on the sensor, it must be matched with a Class 1 device on the DAU. This device is briefly described in section 6.5.

5.3.4 Power Supply

From an electrical point of view, the layout of the sensor circuit is fairly straight forward. The power supply to the sensor and handling of analog values required greater care.

The final sensor design operates on two AA batteries, giving a total of 3V when run in serial. Most modern integrated circuits operate at 3.3V, so the 3V provided was not enough. To compensate for this a pump-charger had to be used, a device capable of bringing a 3V input up to 3.3V. Early prototypes of the sensor used 4AA batteries, giving 6V, but there was some concern that this would too greatly increase the weight of the sensor, possibly past the holding point of the magnets.

Given the analog-to-digital resolution, a very good signal-to-noise ratio was required. This required a precision reference voltage to be used for the analog components (including the accelerometer), and a strict separation between the analog and digital components in both power supply usage and board layout. The accelerometer signals pass through an analog low-pass filter and are buffered to ensure a very low impedance at the analog-to-digital converter. The pump-charger had to be selected to ensure no interference or ribble on the power rails.

Finally, great care was taken to minimize the amount of capacitance in the power supply, as the accelerometer and Bluetooth transceiver had *very* strict power-up requirements to prevent latch-up. In the earliest hardware experimentation phases the accelerometer was tied to a power-supply with a very slow ramp-up time, and this caused the accelerometer to latch and provide incorrect values.

5.3.5 Calibration Procedure

After a sensor arrives from the factory, it is important to calibrate the device to ensure that the values it is returning reflect the desired use of the sensor.

Particularly each sensor must be checked to see how physically skewed the accelerometer is inside the sensor. Ideally the accelerometer face is perfectly parallel with the face of the magnets on the back of the sensor, but small flaws in the manufacturing process (of the sensor) could affect this. The accelerometer is placed on a PCB, and the PCB is screwed into the sensor casing. If the placement of the accelerometer is slightly off, either skewer around the face of it, or not sitting flush on the PCB (possibly due to too much solder under one portion compared to another), then the accelerometer will not reflect the intended orientation. If the screws through the PCB into the sensor casing are not uniformly tight then one corner of the PCB might be raised higher or lower in the casing than the others, causing a similar issue.

The calibration procedure consists of checking each of the X, Y and Z axes against gravity. The sensor is placed on a precisely machined steel structure, and this structure is moved six times, twice per axis.

For example, to measure the X axis, the structure is placed on a perfectly level service such that the X axis *should* be aligned with gravity. The DAU software then asks the sensor to start transmitting values, and the DAU checks the X axis. When aligned with gravity and no motion is occurring, the accelerometer should return exactly $1G$ in one orientation, and $-1G$ in the other. This is repeated for Y and again for Z.

After calibration is completed, the individual values for each axis are stored on the sensor itself. This allows the sensors to be calibrated by any DAU, and then bundled with another DAU for operation. The DAUs simply query the sensors for their calibration data before a Data Acquisition run. Details of the calibration protocol and storage mechanism are given in Appendix A.2.

During the calibration stage an extra operation is performed; the naming of the sensor. Every sensor manufactured gets an unique name. A device name is part of the Bluetooth specification, and is the ASCII string returned by a Bluetooth device responding to a discovery request. The sensors are named in such a way that the DAU can distinguish them as sensors for this system, apart from any other Bluetooth device that might happen to be in range while the DAU is performing a discovery.

5.3.6 Prototype Stages

For completeness, descriptions and images of each of the physical prototype stages are described below, including the reasons for moving between each stage.

Original Prototype

The board shown in figure 5.7 was the first soldered prototype built in the lab (after a few breadboard versions). The PCB was built by hand, and the soldering was also all done by hand.

The Bluetooth chip used here was a BlueRadios Inc. surface-mount device, and the processor was a PIC18F22423. The final product uses the 18F2523 to gain the additional data memory, as described in section 5.3.3.

This prototype included two physical buttons and an On/Off switch, seen in the lower right corner of the image. One of the physical buttons could be held on power-up to put the processor into Upgrade mode, so new software could be uploaded. The other physical button was used to perform a soft-reset on the processor. The final version of the sensor has neither of these buttons, which will be described later.

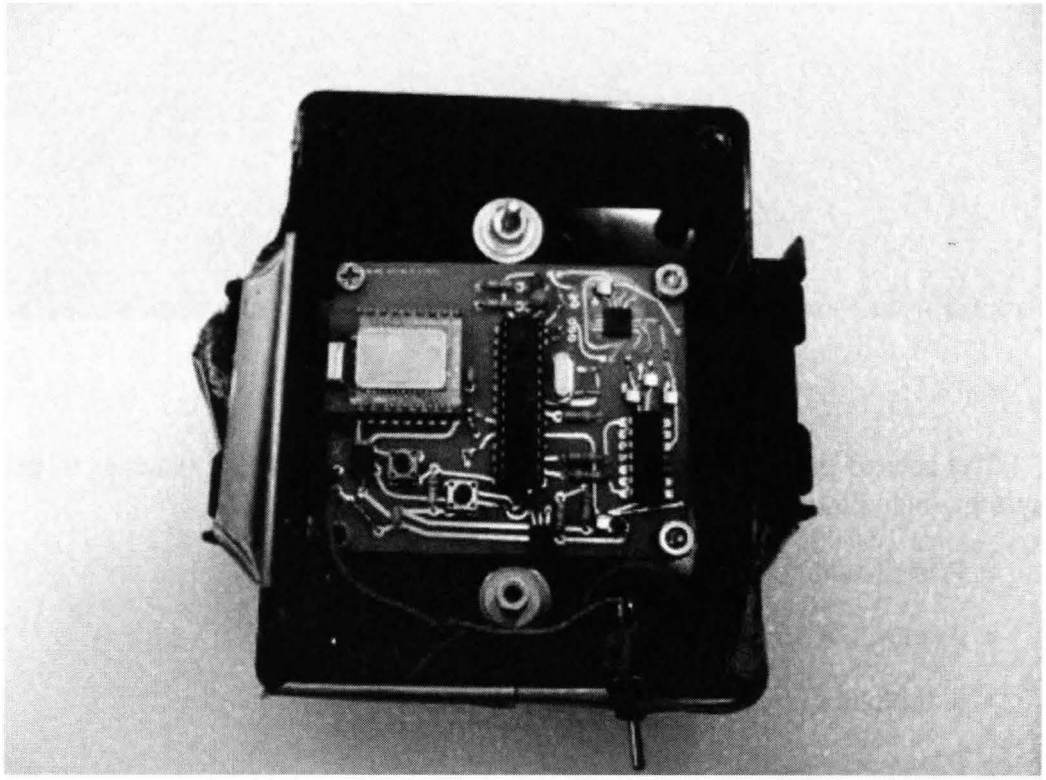

Figure 5.7: Original Prototype Board

Second Prototype

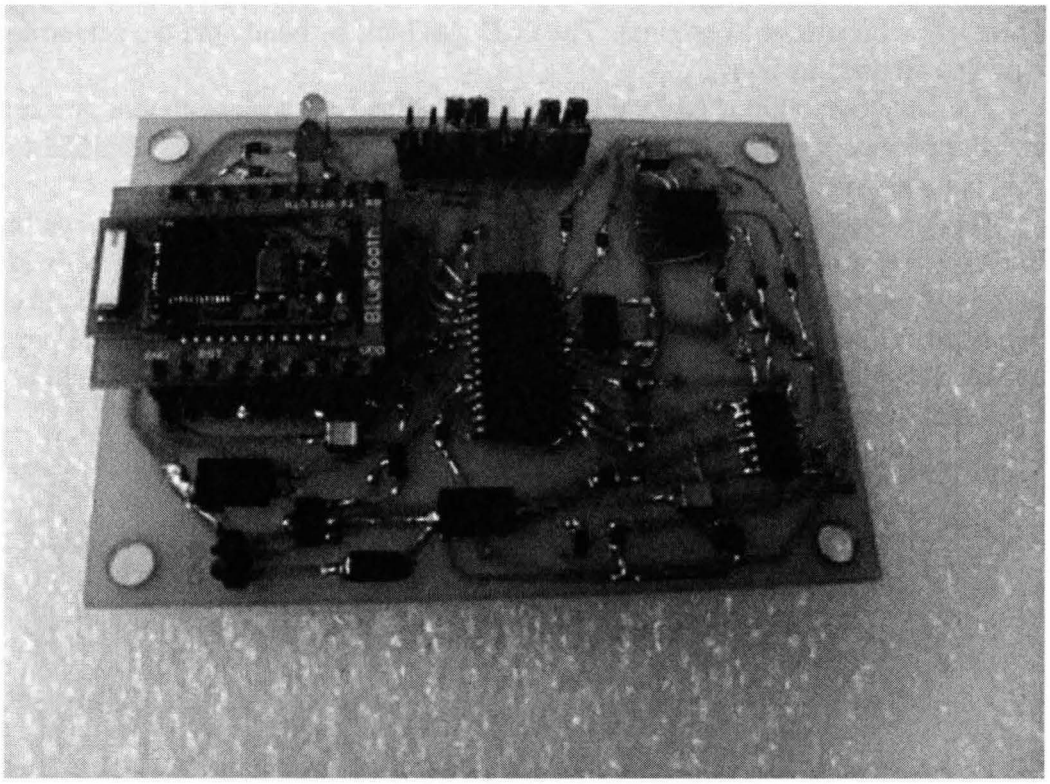

Figure 5.8: Second Prototype Board

The second prototype board (figure 5.8) introduced multiple changes to the system, including:

- Professionally manufactured PCB
- Removal of physical buttons
- Addition of header ports
- Processor swap to 18F2523
- Separation of power supplies
- New Bluetooth transceiver

While the previous PCB had been laid out and etched by hand, this prototype was the first board to be automatically laid out by a software process and built by a professional manufacturer. This allowed the board to be made much smaller, as the manufacturer was capable of building two-sided PCBs, and the software-designed layout is optimized for board size.

The physical buttons were removed early on. The reason was to reduce as much as possible any moving parts on the board, in case long term use of the sensors on high G machinery could wear out the parts. This prompted the move to the header port seen at the top of the image. The header allowed the full functionality of the physical buttons (except for power), but with jumpers instead of buttons. In addition, the header port allowed for in-circuit programming of the 18F2523.

The previous prototype used a Dual In-Line Package (DIP) version of the processor, meaning the processor had physical legs on it to attach to the board, and could easily be removed. Removing the processor from the previous prototype was a common scenario, as the bootloader (section 5.2.2) can only be modified by directly attaching the processor to a special programmer tied to a desktop computer, through a process called In-Circuit Serial Programming [48].

The processor on this new board switched from a DIP version to Surface Mount Device (SMD) version, allowing for a much smaller physical footprint. SMD chips are much smaller than DIP chips, but they must be soldered directly to the board, making it very difficult to constantly remove and reattach them. Unfortunately bootloader changes were still being made at this point, so a method was needed for updating the software without removing the chip. The headers allowed for this. A special cable was manufactured in the lab with a matching header on one end, and a socket that could be used with the programmer on the other. This permitted the in-circuit programming necessary to make bootloader software changes.

This prototype was also the first to employ the separated power supplies described in section 5.3.4. A large amount of noise was being seen on the accelerometer lines in the previous prototype and it was suspected that shared power lines were the cause of this. Moving to separate supplies on this prototype reduced the noise to negligible levels.

Finally this prototype introduced a completely different Bluetooth transceiver, the Philips Semiconductors BGB203. This change was not made for technical reasons, instead it was simply a matter of no longer being able to source the previous transceiver from the supplier. Unfortunately, this change caused a multitude of unforeseen issues, including much more difficult configuration of the transceiver for a new sensor, vastly lower power-output (as described in section 5.3.3 and data

buffering problems as described in section 5.2.6.

First Manufactured Prototype

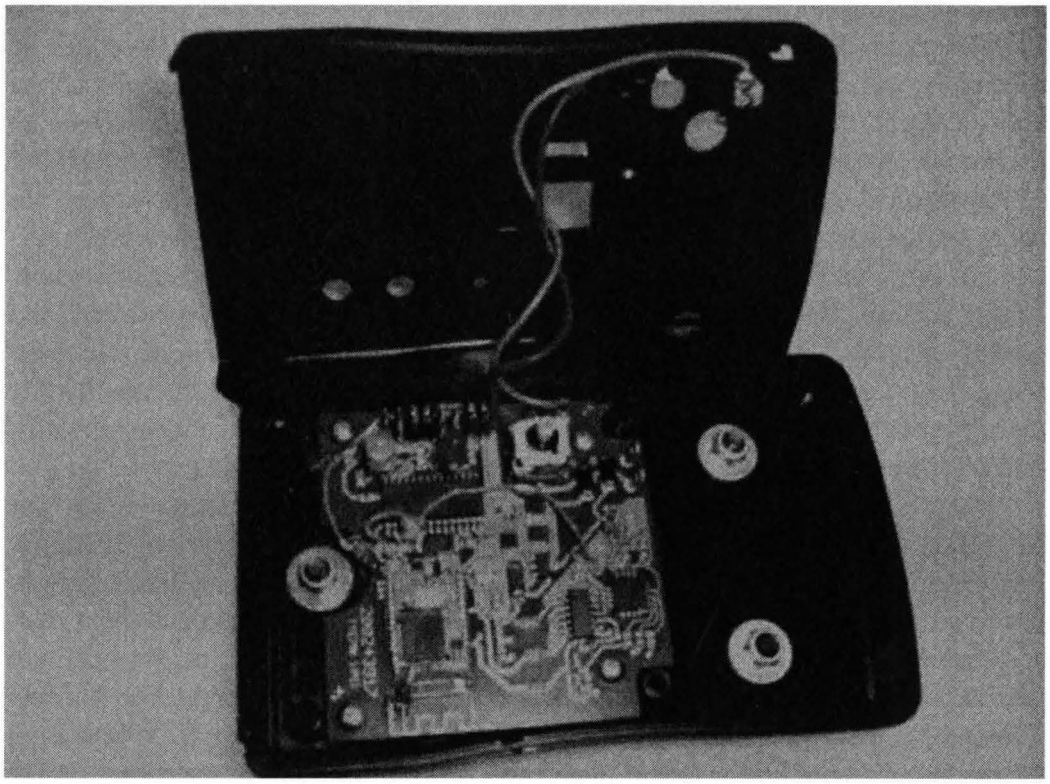

Figure 5.9: First Manufactured Prototype Board

Figures 5.9 and 5.10 show the first fully manufactured board and enclosure, respectively. As the manufacturer is different than the one who designed the previous PCB, a new PCB layout was created. A push-down On/Off button is used on this board, seen attached to the board in the top right of figure 5.9.

This is also the first prototype board to make use of the pump-charger 5.3.4. There was some concern as to whether or not the circuit layout for the pump-charger would be correct, as it was impossible to test it in the lab. The only pump-chargers available that met our desired electrical characteristics were physically too small to solder by hand. Fortunately the circuit was designed correctly and the pump-charger behaved properly.

Figure 5.10: First Manufactured Enclosure

Final Manufactured Device

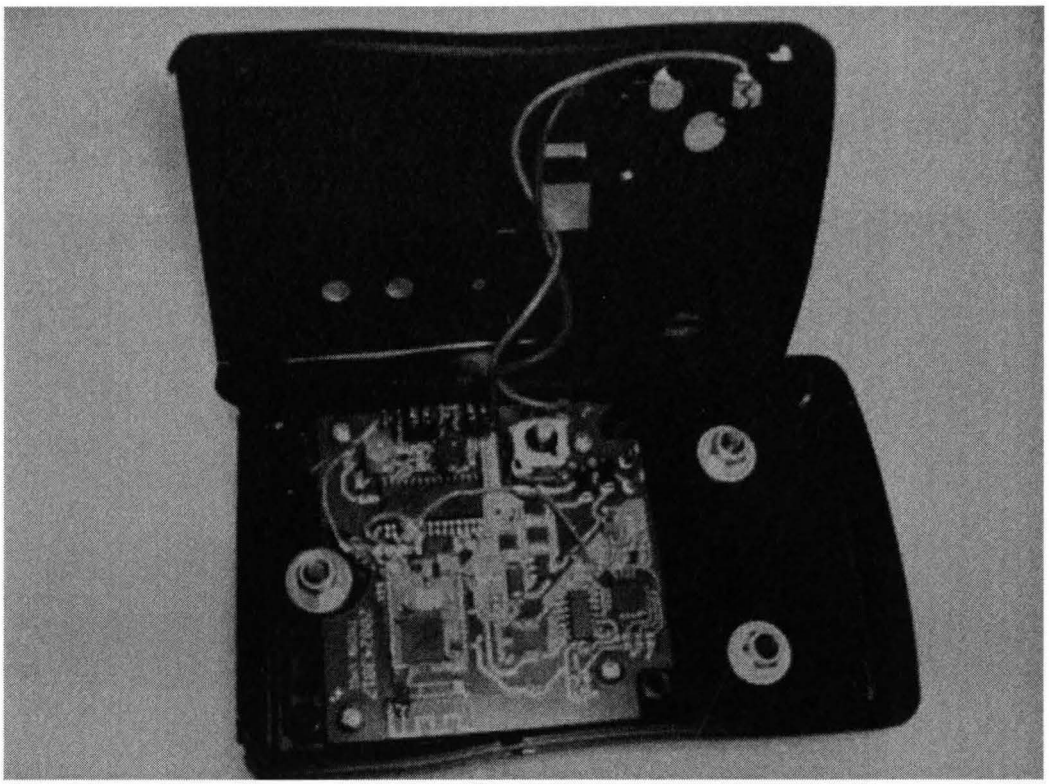

Figure 5.11: Final Manufactured Prototype Board

The final board and enclosure prototypes are shown in figures 5.11 and 5.12. These exact versions are now being produced and have already been distributed worldwide.

Two main changes took place between the previous prototype and this one. First, the On/Off button was changed away from the spring-loaded variety previously used to a rocking toggle switch. This was to alleviate previously mentioned concerns about the durability of moving physical components. While the rocking toggle switch is also a moving component, it is much sturdier.

Second, the Bluetooth transceiver was switched *back* to the one used in the first prototype. After the previous prototypes were built, the low-power issues of the Philips transceiver were discovered. While all the other issues with the Philips device (complex configuration, buffer issues) were already solved with software,

Figure 5.12: Final Manufactured Enclosure

nothing could be done about the power. Fortunately a new supplier was found that could provide the original BlueRadios Inc. Bluetooth chip.

Chapter 6

DAU and Vibration Location Detection Tool Software Designs

Apart from the software running on the sensors, described in the previous chapter, two additional software systems were designed and built. The first is the software running on the DAU for control of the sensors and basic vibration analysis (along with its sister software running on the PC for basic vibration analysis). The second is the frequency/vibration location identification tool mentioned in section 4.5.

This chapter will describe the designs of both software components, starting with the DAU software and followed by the vibration location tool.

6.1 DAU Software

The sensors described in Chapter 5 make up only half of the designed system, as the sensors need to be under the control of a DAU to perform any actions.

The DAU is responsible for controlling the sensors, and is the technician's interface to the system. The only physical interface on the sensors themselves is the On/Off button and the LEDs, after they have been powered the DAU is responsible for configuring the sensors, instructing them when to start collecting data, receiving and recording the collected data, and performing calculations on this data to present the technician with real-time numerical and graphical results of the operation of the vibrating screen.

The DAU is comprised of the hardware platform it runs on and the VA software written for this research. The hardware of the DAU was not designed for this

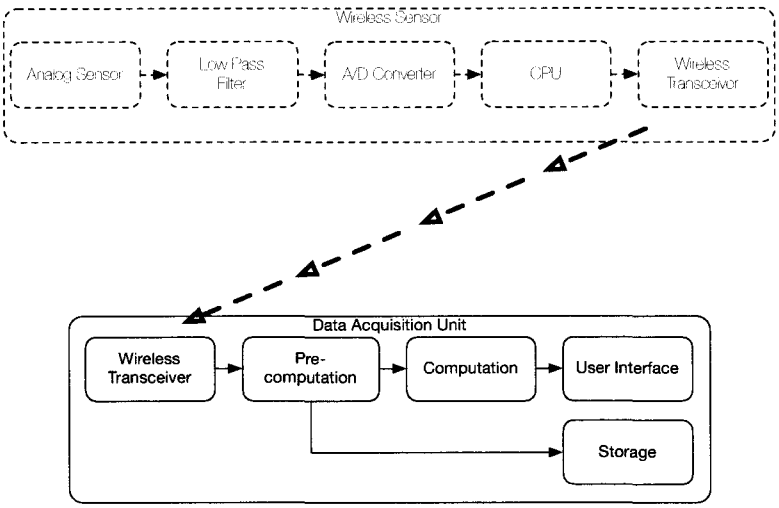

Figure 6.1: DAU Components

project in particular, but the software was.

While not strictly part of the portable DAU that a technician brings to a site, a version of the software running on the DAU is available for regular desktop/laptop computers, which provides some additional post-processing functionality not currently present in the DAU itself. These extra desktop features include ellipse and phase computations, which were too computationally expensive for the DAU, as well as report generation functionality. The report generation requires a large amount of screen space to do properly, so it was left off of the DAU.

The general structure of the software for a wireless sensor system will be presented. While components specific to VA will be described, the overall structure is applicable to any wireless sensor system.

For the purposes of the user interface, a constant iterative cycle took place with a variety of industry technicians. Discussion sessions were commonplace, where the technicians would give their feedback concerning new changes to the system. No user interface feature was ever considered final until a majority of the consulting technicians approved it.

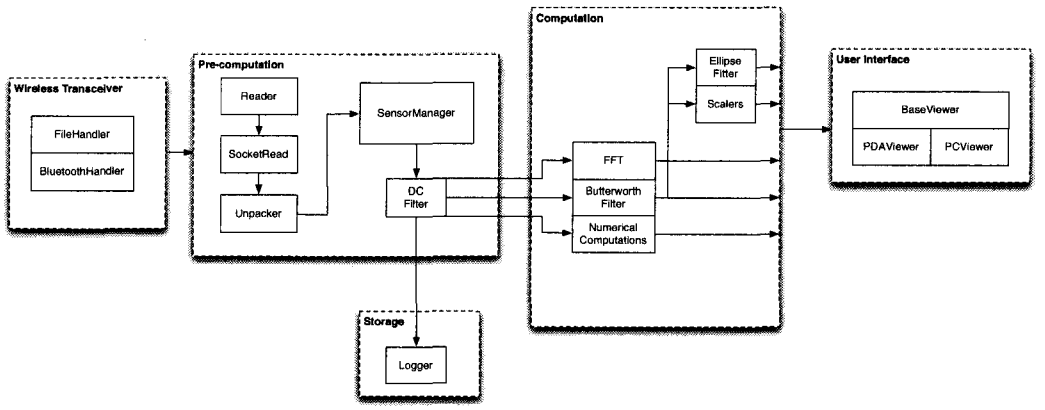

Figure 6.2: Detailed System Components; Arrows show data flow

6.2 Design of the Data Acquisition System

Figure 6.2 shows a more detailed version of the system components graph from figure 6.1. Each of the components from the original figure are highlighted, illustrating the individual pieces that make up these components.

Other modules are part of the system which are not shown here, but they are much more system specific. They can be seen later in the Uses Hierarchy of figure 6.3 as well as in the Module Guide.

The software application written for the DAU is a custom piece of software designed specifically for this project. It is primarily implemented in the Python programming language, with certain performance critical sections being written in Cython [49]. The software was designed with a touchscreen interface in mind, and particularly such that it is usable with just a finger, not requiring a stylus.

The software has multiple duties and responsibilities to perform during a typical data acquisition session, including:

- Provide a simple-to-use interface which at least 95% of the target audience feels comfortable with
- Connecting to a complete set of wireless sensors in less than 30 seconds
- Configuring each of the sensors
- Implementing the control protocol with the sensors

- Starting data collection
- Recording data from sensors at a sampling rate of at least 500 Hz
- Displaying waveforms, orbits and FFT plots
- Displaying numerical results
- Replaying previously recorded results
- Transferring results off the DAU in a manner that at least 95% of technicians find simple

This section will describe the overall design of the DAU software, focusing on areas that required particular care during the design and implementation.

6.2.1 Problem Description

The five blocks from figure 6.2 show the necessary components when designing a wireless sensor system.

Wireless Transceiver

The first block, Wireless Transceiver, actually encompasses a few things. From a high-level, it is viewed simply as the software necessary to speak to whatever wireless component is part of the system. However, this is a limited definition. Notice one of the requirements from the list above was "Replaying previously recorded results".

After a recording session, a technician will often want to replay that particular session multiple times. One could implement a completely separate code-path capable of reading files, performing computations on values from files, etc. The better solution though is to abstract out the interface for receiving data. In the Wireless Transceiver block are the modules BluetoothHandler and FileHandler. The first knows all the details of communicating with Bluetooth hardware, connecting to the hardware, receiving and transmitting with the hardware. The second, FileHandler, presents the exact same external interface as BluetoothHandler. But instead of communicating with hardware, it knows how to read the stored data files from a recording session.

Using this shared interface, the Pre-computation block does not need to concern itself with where the data is coming from. All the details of reading from

a file are encapsulated in FileHandler, such that all data leaving the Wireless Transceiver block is identical in form, no matter what the source was.

In addition, by abstracting out the details of the hardware interface, this allows for future extensions to change the hardware interface, without having to adjust any other part of the software. If a future version of the sensor uses a different wireless technology, it will only require a new Handler, limiting the changes necessary to the rest of the code.

Even during development this proved beneficial. When first developing the hardware for this project, a wired connection was used between the sensors and the DAU. This was simply for testing purposes, before the Bluetooth was ready. When it was time to move to the Bluetooth, only a new Handler was required, it did not affect the other components of the system.

Pre-Computation

As shown above, a Pre-computation block should primarily be responsible for two things:

1. Detecting the presence of, and reading, new data
2. Collecting and preparing this data for the Computation block

The three left-most modules, Reader, SocketRead and Unpacker are responsible for the first of these. Reader abstracts out the method for detecting *when* new data is available. It then informs SocketRead which is responsible for pulling out the data. Unpacker is responsible for decoding that data, given whatever encoding scheme is chosen. Recall figure 5.6 for the scheme used in this system.

This separation into the three modules makes sense given the likely changes that might occur in the future. The three logical steps of this part of pre-computation are

1. Detecting the presence of new data
2. Reading this new data
3. Interpreting the data

For example, the best way to detect the presence of new data is highly dependent on operating systems and networking techniques. A system developed for Linux might want to use the `epoll` interface, while one for OS X might prefer

`kqueue`. And both of those are assuming a standard socket is used. Changing the communication mechanism would change the detection method.

Reading the data is also dependent on the chosen networking techniques. Standard sockets in most operating systems support a `read()` function, with a standardized interface. The use of something other than a socket would change this.

Finally, interpretation of the data is wholly dependent on the protocol created for transmitting the data from the sensors.

The second stage of the Pre-computation block has the interpreted data going to some kind of SensorManager, which in turn sends the data for DC filtering. The act of DC filtering is dependent on a VA situation where gravity is present, but many systems would employ some kind of early stage filtering here.

The SensorManagers themselves represent a mechanism for coordinating between the low-level interfaces to the datastreams, and the higher-level computations and user interfaces. The true interaction in the developed system is slightly more complex than the diagram above suggests.

Computation

The Computation block is where most of the mathematical computation and analysis will take place. It is responsibly for performing the computations necessary to display useful results to a user in the User Interface block.

The internal modules shown above are fairly specific to VA, but this is not surprising. The actual computations that take place will always be domain specific.

The computation block shows all the internal modules passing data through the Computation boundary, as each of these modules generates data that can be displayed in the User Interface.

User Interface

The modules in the User Interface block are drawn to represent their tight interdependency. BaseViewer provides all basic interfaces services, while the PDAViewer and PCViewer modules provide device specific display customizations.

The goal of separating the modules in such a way is to make it as easy as possible to add new "views", new ways for a user to interact with the system. For the VA system developed, this includes the DAU as well as standard laptops and desktops.

The rest of this chapter will present how these blocks were actually implemented in the system we designed. They will be explored in further detail, presenting problems and solutions that were encountered during the design and development.

6.2.2 Module Guide

This section presents the Module Guide for the system, as described by Parnas [50]. Specifically,

> It defines the responsibilities of each of the modules by describing the design decisions that will be hidden (encapsulated) by that module (its secrets).

More modules are present here than in figure 6.2, representing more of the specific details needed to implement the actual system.

Name	BaseViewer
Service	Main entry point of software, controls GUI interactions
Secret	GUI implementation data structures

Table 6.1: Module: BaseViewer

Name	PCViewer
Service	Provides desktop-specific GUI features
Secret	Algorithms for desktop screen layout

Table 6.2: Module: PCViewer

Name	PDAViewer
Service	Provides PDA-specific GUI features
Secret	Algorithms for PDA screen-size layout

Table 6.3: Module: PDAViewer

Name	SensorManager
Service	Processing and short-term storage of data samples
Secret	Processing algorithms and internal storage data structures

Table 6.4: Module: SensorManager

Name	FFT
Service	Performs frequency domain calculations
Secret	FFT and interpolation related algorithms

Table 6.5: Module: FFT

Name	CircularQueue
Service	Provides a thread-safe circular queue data structure
Secret	Algorithms and data structures to implement the circular queue

Table 6.6: Module: CircularQueue

Name	Logger
Service	Writes data samples to a log file, and reading from a log file
Secret	Data structure of log files

Table 6.7: Module: Logger

Name	Calibration
Service	Transforms data samples based on stored calibration data
Secret	Algorithm for transformation

Table 6.8: Module: Calibration

Name	DC Filter
Service	Implements a DC filter to remove DC components from a signal
Secret	DC Filter coefficients

Table 6.9: Module: DC Filter

Name	Butterworth Filter
Service	Implements an adaptive bandpass Butterworth filter
Secret	Coefficient calculation

Table 6.10: Module: Butterworth Filter

Name	Handler
Service	Base class providing low-level communications with data sources
Secret	Algorithms for communicating with a data source

Table 6.11: Module: Handler

Name	Ellipse Fitter
Service	Calculates eccentricity and ellipse-phase
Secret	Algorithms for calculation

Table 6.12: Module: Ellipse Fitter

Name	Scalers
Service	Responsible for scaling data so it fits on-screen properly
Secret	Algorithms for performing the scaling

Table 6.13: Module: Scalers

Name	Utility Menu
Service	Implements all functionality available in the Utility Menu
Secret	Algorithms and data structures for various functionality

Table 6.14: Module: Utility Menu

Name	Data Export
Service	Services to export recorded data off the DAU
Secret	Algorithms for the particular export implementation

Table 6.15: Module: Data Export

Name	Reader
Service	Provides the main run loop of the communication channels
Secret	Algorithms and techniques for detecting new data and handling it

Table 6.16: Module: Reader

Name	SocketRead
Service	Reads data from a given communication channel
Secret	Algorithms for reading data and tracking packet progress

Table 6.17: Module: SocketRead

Name	Unpacker
Service	Decodes raw stream data into packets
Secret	Communication protocol between the sensors and the DAU

Table 6.18: Module: Unpacker

Name	ReportGeneration
Service	Generates final reports for a particular VA session
Secret	Algorithms and data structures for layout of reports

Table 6.19: Module: ReportGeneration

Name	Numerical Values Computer
Service	Calculates numerical results of incoming data
Secret	Algorithms and data structures for calculation

Table 6.20: Module: Numerical Values Computer

Name	Report Calculations
Service	Performs numerical computations required for final reports
Secret	Algorithms and data structures used to perform calculations

Table 6.21: Module: Report Calculations

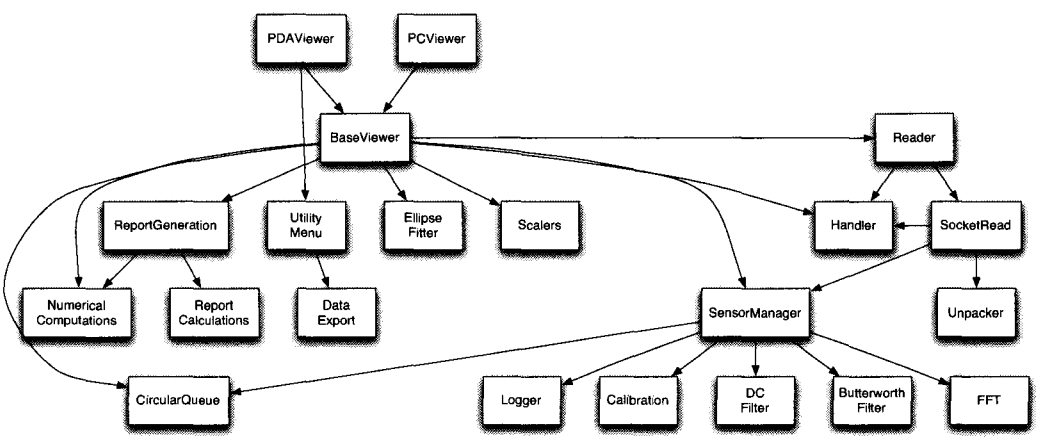

Figure 6.3: Uses Hierarchy

The Uses Hierarchy in figure 6.3 shows how the modules interact with each other. An arrow between modules A and B like $A \rightarrow B$ means that "module A *uses* module B".

6.2.3 Main System Loops

After initial configuration and connection phases with the sensors, the software can be generalized to be performing two main tasks:

1. Collecting and processing data from all connected sensors
2. Displaying processed data on screen

Figure 6.4 is similar to figure 6.3, but now shows two logical groupings, "Display" and "Collect and Process", representing the modules that are employed in the two separate tasks.

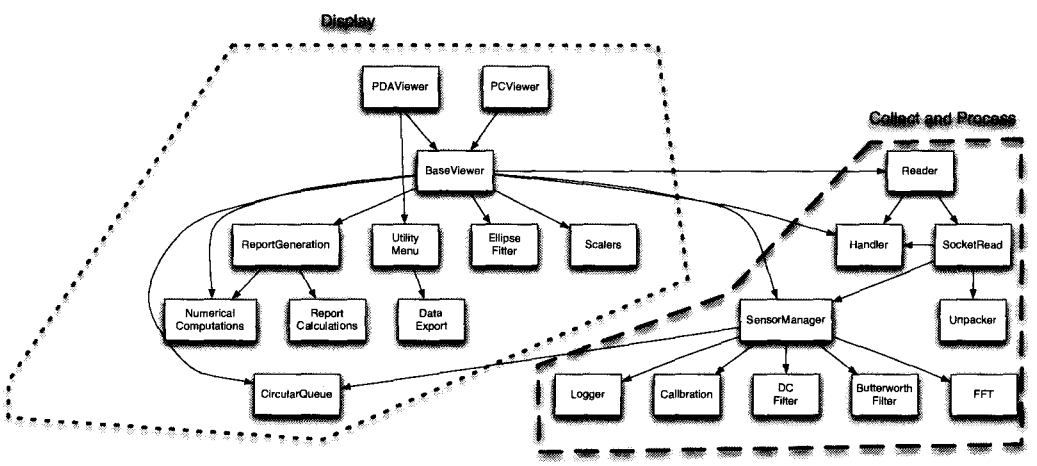

Figure 6.4: Grouped Uses Hierarchy

The "Collect and Process" modules are those that communicate directly with the sensors. The acceleration data is collected and decoded, before being sent to the SensorManager by the SocketRead module. The SensorManager is where filtering happens on the data, and the results are stored into two separate queues, "Filtered Data Queue" and "Raw Data Queue". Unfiltered data is considered to be data that has been transformed for calibration and DC filtered (to remove the

constant gravity component) but not Butterworth filtered. It might be odd to call it "unfiltered" when it has in fact passed through a filter, but the DC filter does not affect the shape of the machine's motion, it simply removes the constant G-force resulting from gravity.

Filtered data is the result of passing this raw data through a Butterworth filter.

The Butterworth was introduced in section 2.6, but in short it implements a bandpass filter, centred around the main operating frequency of the rotating machinery under analysis.

This is also the stage where data logging happens. The only data that is logged is the DC filtered raw data. The filtered data can always be recreated, so there is no point in separately logging it.

This process is shown in figure 6.5 to emphasize the flow of data through this main processing stage.

After placing the filtered and unfiltered data in the appropriate queues, the "Display" process is able to grab the data and display it appropriately.

These queues represent the thread barrier of the software. The "Collect and Process" and "Display" operations run in separate threads, and the only interaction between these threads (during data collection) happens through the queues, as shown in figure 6.6. These are thread safe queues, with Thread 2 acting solely as a producer, and Thread 1 as a consumer, essentially acting as an asynchronous message passing system (the queues are non-blocking, hence asynchronous). Message passing not only makes the interaction between the threads easier to manage, avoiding the complexity of mutexes and semaphores around shared resources, but also adds a level of safety. In software where threads share resources, if one thread crashes then typically the state of the other threads becomes corrupted [51]. In a message passing system a single thread can die without ruining the state of the threads it communicates with. Strictly limiting thread communication to message passing is often called "Erlang style concurrency" [52], after the Erlang programming language.

Why is it that threads are even required, why not use just a single process of control? wxWidgets, the chosen graphical toolkit, essentially requires this by design. Like most modern graphical toolkits, it is an event based system, with its own run loop. When the software starts, wxWidgets takes control of the main run loop of the system, and various events (button presses, key presses, etc.) are registered with the system, along with appropriate event handlers. When the toolkit receives a graphical event notification from the operating system, it checks if any handlers are registered to respond to that event, and if so, calls the appropriate handler function. After the function completes the main loop continues to listen

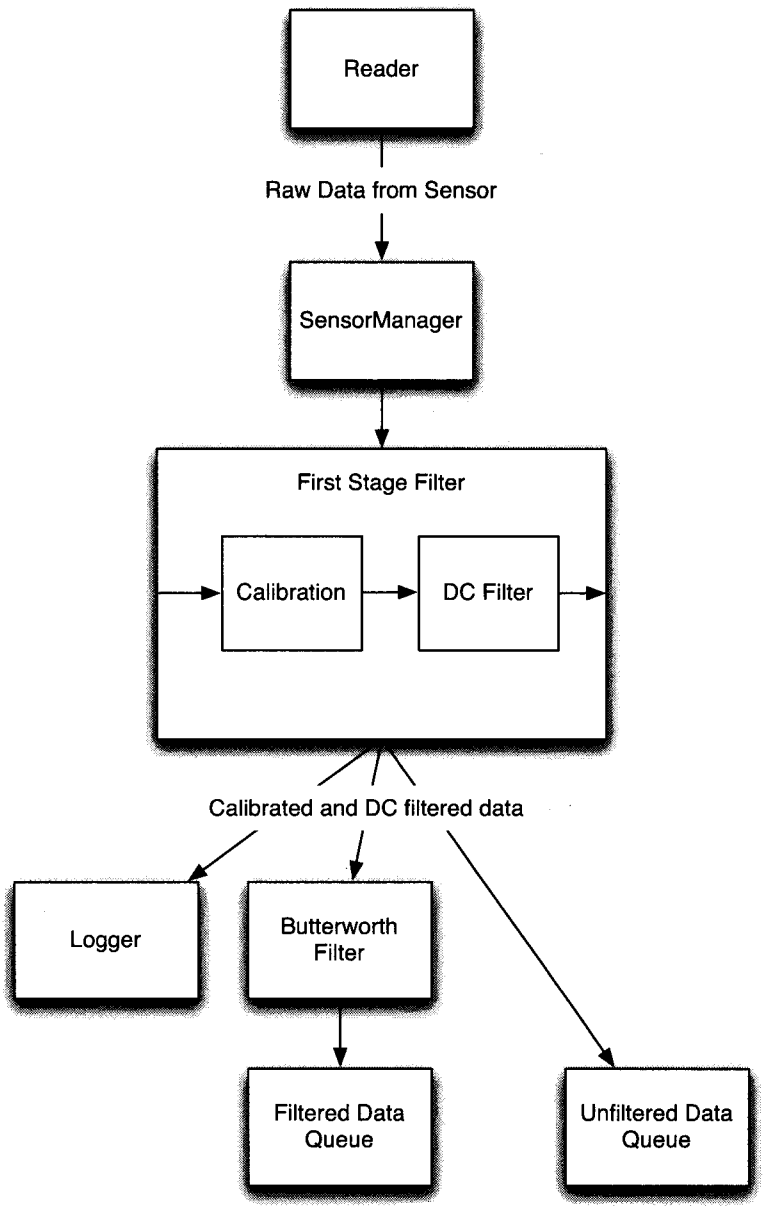

Figure 6.5: Main Loop

for events. This means that none of the DAU code is executed, except in response to graphical events.

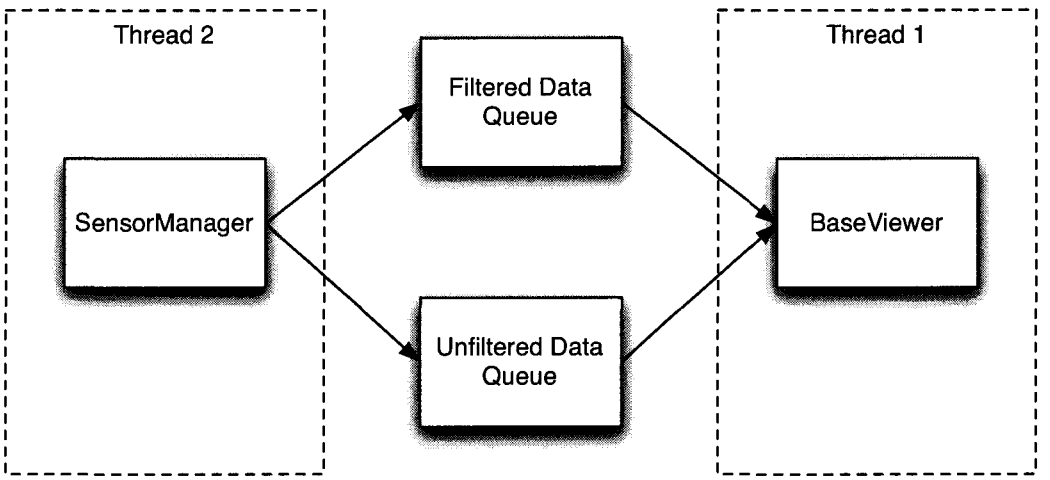

Figure 6.6: Thread Barrier

This is problematic for software that is constantly receiving data from non-graphical sources, such as the data coming in from the sensors. Collection and processing of the sensor data must continuously occur, and this does not fit well with the event-based model of the graphical event run loop. For this reason the collection and processing software is started in a separate thread, so it can run independently of the wxWidgets run loop. This separate thread is then free to do its work, and simply place its results into the "Filtered Data Queue" and "Unfiltered Data Queue".

The "Display" thread runs a low-precision wxWidgets-based timer, which is used to periodically pull data from the two queues and update on-screen graphs and calculated values. An attempt was made to use this timer to run the data collection, but its accuracy was too low, and the collection code was not being called often enough.

6.2.4 BaseViewer

A very important module in the "Display" group, as seen by the number of modules it uses, is BaseViewer. This module implements most of the base classes used for the GUI, and is responsible for registering the event handlers used by the wxWidgets main loop. It is also responsible for creating the SensorManagers, Handlers and the Reader thread, the three main points of control and activity in the "Collect and Process" group.

An important design goal was to make the DAU software usable not only on the PDA, but also to allow versions of it to run on Windows and Mac computers. Not only do these alternate systems provide different operating systems (compared to Linux running on the PDA), but more importantly the look and interaction methods different drastically between the DAU and the desktop computers. The DAU runs at a relatively small 480x640 resolution, while Windows and Mac machines could theoretically be at any resolution. The DAU also expects most input and interaction to happen with a technician's finger, while desktop machines typically expect keyboard and mouse. This requires large buttons and interfaces on the DAU [53], where less pointing precision will be available, but also requires that the on-screen data displays be minimized for the vastly smaller screen space.

To accomplish both goals simultaneously, object-oriented inheritance and composition structures were used. BaseViewer defines the software interfaces to be used in all systems, and implements all logic that would be common between different visual interfaces, but makes no assumptions about screen-size or intended input method. Instead wherever a situation comes up where that information might be required at the BaseViewer level, an abstract method is defined, one that can be overridden and specifically implemented in the appropriate child class. In certain circumstances it made more sense to use composition instead of inheritance, the two styles were intermingled as deemed necessary.

The two modules defining the child classes and appropriate composition objects are PDAViewer and PCViewer. PDAViewer implements the elements necessary for the DAU's user-interface, while PCViewer handles the user-interface for a desktop-style system. No real distinction is made between Windows and Mac system, as wxWidgets automatically handles drawing operating-system-specific widgets and interface objects.

Note only do these modules implement the functionality that BaseViewer requires, but there are certain functionality that exist only in the DAU software or the desktop software, not in both. These platform-specific functions can be found in PDAViewer and PCViewer.

6.2.5 PDAViewer

The main tasks of PDAViewer are to implement the sub-classes and composition components required by BaseViewer to run the software on the DAU. This consists of laying out buttons and other widgets on-screen, and implementing a few bits of logic that are dependent on the platform.

One feature that is only available on the DAU is the "Utility Menu" 6.7. This

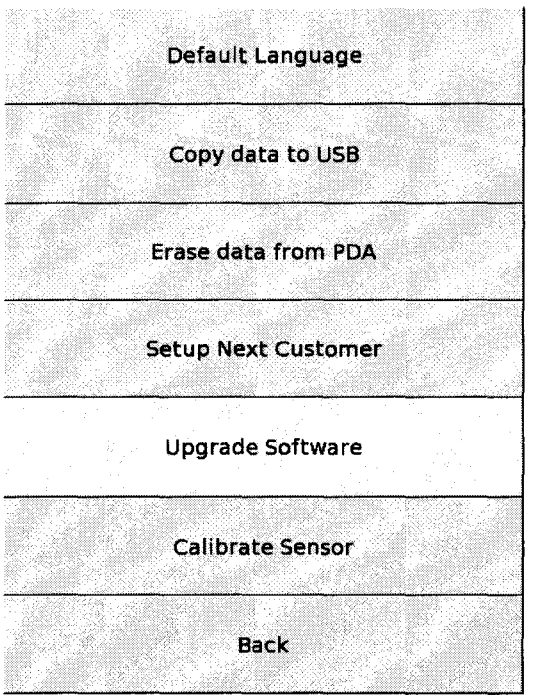

Figure 6.7: Utility Menu

provides functionality specific to operating the DAU. The DAU is intended to be used as a black box, the users should not need to know that it operates on Linux, should not need to understand how to perform software upgrades on a Linux system, or know how to work with the Linux filesystem. The Utility Menu provides an abstraction around all of this functionality, in a way consistent with the typical use of the DAU.

PDAViewer is responsible for inserting access to this functionality into the user-interface, and communicating with the "Utility Menu" module which implements the actual mechanics of these operations.

Figure 6.8 shows the main running screen of the PDA, with eight sensors running and transmitting data from a vibrating screen. This shows the unfiltered view of the data, before recording has begun.

An important consideration when developing PDAViewer was the overall usability of the system on the small screen employed by the DAU. As previously mentioned, the DAU is often used in harsh environments. It is not unusual to have very little light to work with, and for the area to be surrounded in dust.

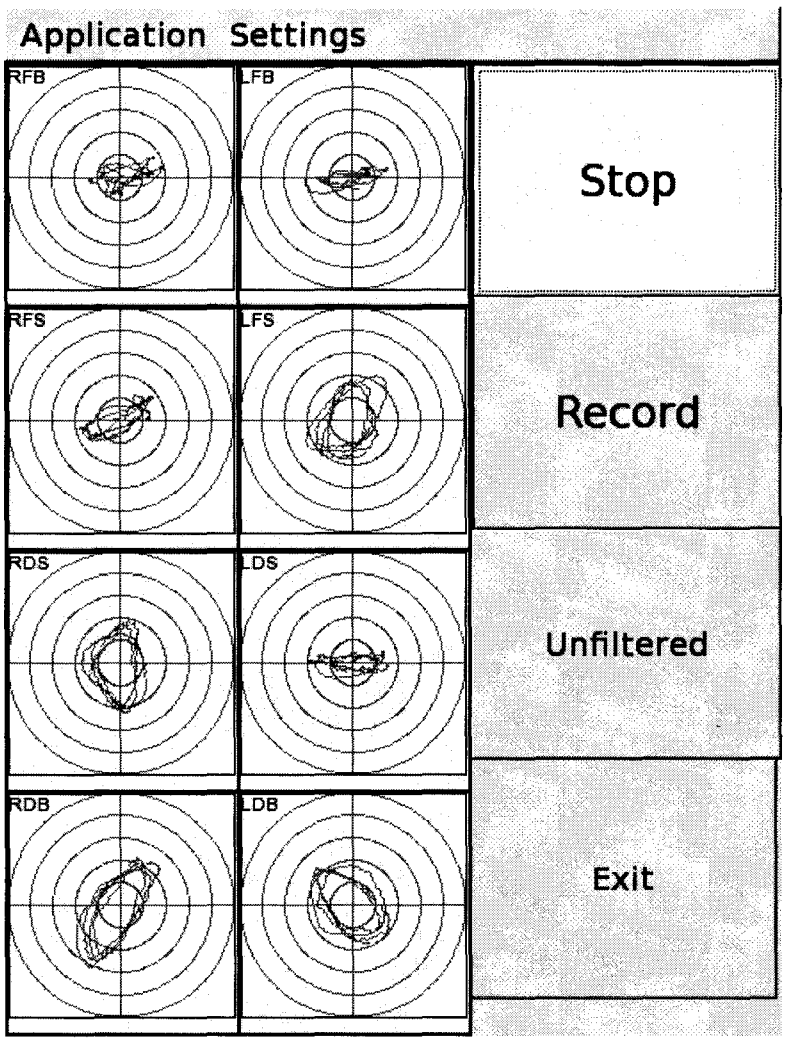

Figure 6.8: Main Loop

Early versions of the DAU software had many overly-small interface elements, which were often hard to read, and required the use of a stylus to operate. Usability tests with industry technicians showed that this was unacceptable. Every technician polled said that it would be too hard to read the interface in a real-life situation, and that reliance on a stylus was a bad idea. Because of the nature of the environments, there is a good chance that a dropped sytlus would never be found again.

The solution to this was a complete redesign of the interface. Everything was rethough and resized, such that all elements could easily be pressed with a finger, rather than a stylus. This necessitated a size increase of all elements. This size increase brought everything to a level such that not a single technician we polled said they would have a problem with it.

Another interface element that was developed with real technicians in mind was the data entry system. Before beginning a test, the technician is responsible for entering various pieces of data associated with the screen: customer name, manufacturer name, serial number, number of bearings, date, etc. Technicians found this incredibly tedious in early prototypes, and would often leave out information, just to save time.

A feature was then added to simplify this. Instead of having to enter all the information every time, the system would default to the information from the last measured screen. Technicians often have multiple screens at the same customer site that they must analyze, and what we found was that the only information that usually needed to change was the serial number. By presenting the information from the last screen, the technicians then usually just had to change the serial number.

In addition, a Utility Menu item was added to let the technician enter this default information ahead of time. Before heading to the customer site, they could fill in the information that would be common between all the screens, saving themselves some time at the customer site.

These are just some of the decisions and features that were required on the DAU to aid technicians in the field, and it was with the feedback of these technicians that led to the features.

6.2.6 Reader

The Reader module is responsible for checking which sensors have new data for the system, and appropriately dealing with the sensors.

As mentioned above, the software runs two main threads. The "Collect and Process" thread's main loop resides in Reader. This main loop continuously checks each sensor for new data. This is done via the standard POSIX socket interface [54]. The main loop collects all the active sockets from the Handler module and every time through the loop checks the status of each of the sockets using the standard `select()` function. This function tells Reader whether each socket has any data ready, or if an error has occurred on the socket.

It should be mentioned that `select()` is simply an implementation detail. `epoll()` or another similar interface could be used instead, depending on operating system availability. `select()` was chosen because it is available on all of the current target operating systems. Each operating system tends to provide their own particular method of doing something similar, usually with much higher performance. If it is ever found that the use of `select()` has become a bottleneck, then operating system-specific implementations could be used.

If no data is available for a given socket, then a special "timeout" variable is incremented for that socket. After this timeout variable reaches a pre-determined threshold, it is assumed that for some reason the connection to the sensor has been lost, and the socket is marked as dead.

When a socket does have data available, its timeout variable is first reset to 0, and then the socket is passed off to the SocketRead module for reading and processing. Despite its name, the Reader module does not actually read data from the sockets, it is strictly responsible for detecting when new data is available, and looking for dead connections.

When a socket is marked as dead, this information is propagated to the "Display" thread via the Message Queue. This provides a means for the technician using the DAU to be informed that the connection was closed, allowing them to ask for the connection to be re-initiated. Because of this, a secondary job of the Reader main loop is to check for the availability of newly started sockets. Information about new sockets comes to the Reader via the same Message Queue, after which it must add that socket to the collection of sockets it is already checking each time through the loop. Because this is a very rare occurrence, the Reader module does not check for new sockets every time through the loop. Instead it checks every 20 times through the loop.

This value of 20 was determined through experimentation. Accessing the Message Queue is a reasonably expensive operation, as it implements a *two*-way thread-safe queue, which automatically performs all the locking and unlocking operations needed every time it is accessed. Checking this queue too often caused performance to degrade. Checking every 20 times through the loop was found to have a negligible performance impact, while still being often enough that a user will not have to wait too long in-between asking for the connection to be re-established and Reader actually checking that connection for new data.

Why would a connection die in the first place? There are multiple reasons this could occur, including:

- Battery dies on a sensor during operation

- Technician moves the DAU out of range of a sensor
- Sensor is accidentally powered down

It is not uncommon for the DAU to be moved out of range. Most technicians move around the vibrating screen while performing VA, to see if they can visually identify potential problems with the screen that might be responsible for the data they're seeing on screen. This can put the majority of the screen between the technician and the sensor, forcing the Bluetooth signal to pass through a massive steel structure. Even with the high-power Bluetooth devices used in the system, this can still cause the signal to drop.

This was happening with just enough regularity that a system for re-establishing connections during a run was required, as opposed to early versions of the software that required the entire VA procedure to be restarted.

When at a site, technicians are often tasked with a large number of tests to run. The early versions of the software required that the whole system be started over again if a single sensor connection was lost, and this simply cost the user too much time. While it was no problem in a lab environment, where the sensor were never too far from the DAU and the batteries were always freshly charged, it was a serious problem in the real-world.

This necessitated new components capable of notifying the user that a connection was lost, presenting a new interface for re-establishing the connection, and attempting a new connection with the disconnected sensor. These would have been unnecessary in a system meant simply for lab use, but completely necessary on a real-world system such as this.

6.2.7 SocketRead

Whenever the Reader module sees that a particular sensor has data available to be read, it tells the SocketRead module to actually do the read. The definition of "data available to be read" is that a sensor has transmitted data to the DAU, and that data has passed through the communication stacks of the DAU to a point such that it is available for reading into userspace.

The primary argument that is passed to the SocketRead module is the Handler instance for the particular sensor. The SocketRead module does not speak directly to the low-level communication routines, instead using higher-level abstractions provided by the Handler module.

The first task of the SocketRead module is to actually read the bytes of data that Reader was told were available. This is done using an interface provided by the Handler.

Once the data has been read, it is appended to any "unused" data from the last time this particular sensor was passed to SocketRead.

It is important to remember (as mentioned in section 5.2.10) that a full packet takes 10 sample periods on the sensor to fully transmit, but that the sensor does not wait until the full packet is available before it actually begins passing the data to the Bluetooth transceiver for transmit to the DAU. The Bluetooth chip has its own buffers and networking protocols that will determine when particular bytes are actually transmitted, but the general situation is that the Bluetooth chip is simply transmitting a stream of bytes. Even when the data does arrive at the DAU, the Linux kernel and Bluetooth stacks will make their own determinations as to when to pass received data up to userspace (such that `select()` will see that data is available).

For all of these reasons, SocketRead will typically not receive multiple full packets when it reads the data from the socket. Instead, it is much more likely that the beginning of the data read off will represent the end of the previous packet, then there will be one or two full packets, followed by the beginning of another packet.

This is illustrated in figure 6.9, which shows a situation in which $3+60n+2$ bytes happened to be available and read in. The first 3 bytes represent the end of a packet that had only partially arrived at the DAU during the previous read. The next $60n$ bytes represent n full packets that arrived during this read. The final two bytes are the header bytes for the next packet, the rest of which is not yet available to be read.

This clearly shows why on a read in SocketRead any "unused" data (the final two bytes in figure 6.9) must be stored, and the data on the next read must be appended to this unused data.

The Handler instance for the sensor will store this data but it is the responsibility of SocketRead to perform the logic necessary to append data and decide what new data must be held over until the next read.

After determining how many full packets are available for processing, the raw data representing those packets is passed to the Unpacker module where it is decoded and turned into accelerometer readings.

Finally, the accelerometer readings are passed to the appropriate SensorManager instance for the sensor, for filtering and all subsequent processing.

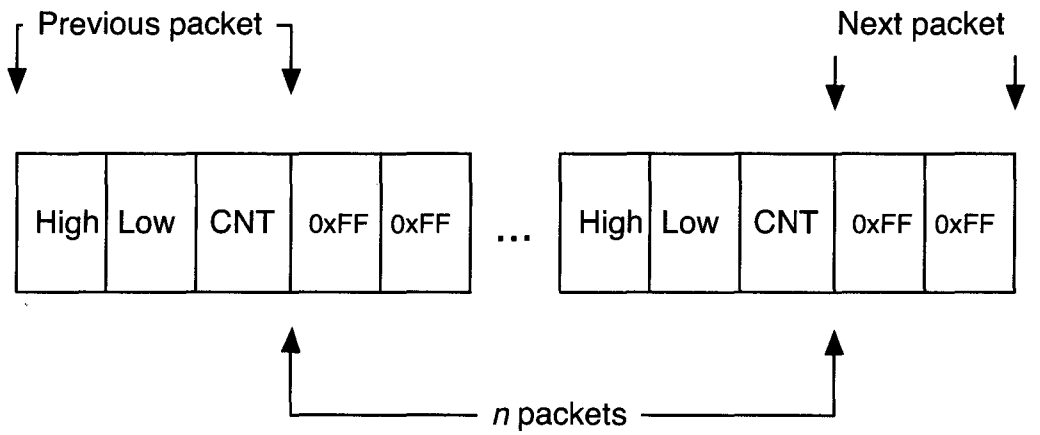

Figure 6.9: Packet Fragments

6.2.8 Handler

The Handler module is responsible for controlling access to the sensors, or in cases of replaying data from a file, access to the files themselves. This is where the low-level protocols described in section 5.2.9 are actually implemented on the DAU. Whether communicating with live sensors or simulating a sensor by replaying data from a file, the Handler module provides a consistent interface to the other modules in the program for accessing these services, without those modules having to know the low-level details of communicating with a sensor or reading from a data file.

The Handler is also responsible for creating the sockets used for communication with the sensors. The implementation details of this are specific to the particular Bluetooth libraries used in the software, but the external interface is consistent, so if the Bluetooth library used happens to change, the calling modules will not have to change their interaction with Handler.

The technical aspects of reading data from a file are naturally much simpler than connecting to a remote Bluetooth device, but the interface remains the same. A desire throughout the rest of the software was, as much as possible, to write the modules such that they did not need to be aware of whether or not the data was coming from a running sensor or from a file. Wrapping access to data files with the Handler module ensures that the rest of the software only needs to know how to interact with the Handler module, and not be concerned with the actual source

of the data.

There are currently implementations for Bluetooth sensors and local data files, but if more data sources were to be added in the future (hard-wired sensors, network-connected databases storing global results, etc.), adding support would be a trivial task, and the rest of the software would not need to be modified.

Another important role for the Handler, at least in the Bluetooth case, is to track the packet counter variable and temporarily store unprocessed data.

The packet counter shown in figure 5.6 must be tracked in the DAU, to ensure that the packets are arriving in the correct order, and no data corruption is occurring. The Handler module tracks the last packet counter value for each sensor. The process of checking that the next incoming value is correct is handled elsewhere, but it is important to note that the value itself is stored in the Handler. The Handler is intended to encapsulate the complete set of control routines and state for each connection, including the packet counter value.

The Handler is also responsible for storing any unprocessed data. The Reader loop only checks that *some* data is available from a particular sensor, not necessarily that an entire 63-byte packet is available. SocketRead pulls all the data that is available from the sensor, and proceeds to process all the complete packets that are available. If only part of a packet has arrived, then that partial packet must be stored until the next check for data occurs.

6.2.9 FFT

The FFT module is responsible for performing the key FFT operations in the system. This includes performing an FFT over every 2048 samples, calculating the fundamental frequency and RPM, and refining these values with a polynomial interpolation scheme.

In section 2.7.2 the DFT interpolation was discussed. One question left open was whether or not it was worthwhile to perform both polynomial interpolation *and* DFT windowing.

The system was designed to perform 2048 point FFTs. At a sampling rate of 500Hz, this equates to starting an FFT computation roughly every 2 seconds. One must remember that the FFT is calculated for each of the three axes and for up to eight sensors. A 2048 point FFT would require a 2048 DFT window, so with eight sensors, each requiring three full FFTs, this would equate to $2048 \cdot 24 = 49152$ extra multiplications per complete round of FFT computations.

These extra multiplications were deemed too expensive to run on the DAU, though if the extra accuracy is deemed important enough, techniques are avail-

able to modify the source code of the FFT function such that the multiplications essentially happen for free. The complexity to modify the FFT routines was not thought to be worth the effort at this time.

For any VA system, this question would have to be asked again. Is the extra accuracy provided by DFT windowing worth the extra computations, and more plainly, does the VA system even have enough computational resources to add DFT windowing to the rest of its calculations.

6.2.10 Butterworth Filter

While software such as MATLAB and Numpy can generate the necessary IIR coefficients for a Butterworth filter of pre-designed specifications using their `butter` function, and those coefficients can be passed to the `filter` or `lfilter` functions (MATLAB and Numpy, respectively) to perform the actual filtering, the software on the DAU does not make use of these features.

The `butter` function requires the desired centre frequency of the bandpass Butterworth filter as part of the filter's specification, but the DAU does not know this centre frequency ahead of time. Instead, the desired centre frequency of the filter is calculated based on the actual operating frequency of the vibrating screen determined during RPM calculation.

This frequency is then passed to a custom function implementing a fourth-order Butterworth filter, where the coefficients are generated during run-time. Because of the high computational cost of calculating the coefficients, this operation only takes place whenever the centre-frequency has changed past some tolerance level.

The filter then implements a high performance calculation phase, taking in samples in chunks with length a multiple of 4, allowing for fine-tuned loop unrolling[55] based around a Butterworth filter of order 4.

The knowledge that the Butterworth filter will be order 4 allows for these optimizations. With eight sensors running, the DAU will have to have 24 Butterworth filters running simultaneously, making it one of the most computationally expensive components in the entire system. Passing samples in chunks reduces cache misses, and using a hand-rolled filter evaluation routine made specifically for this purpose gave much higher performance than the `filter` or `lfilter` functions.

To show the correctness of our implementation, the coefficients were generated (for an operating frequency of 14Hz) and then used with the `freqz` [56] function to produce the frequency response. The overall frequency response is shown in figure 6.10, and the magnitude response alone in figure 6.11. These show the

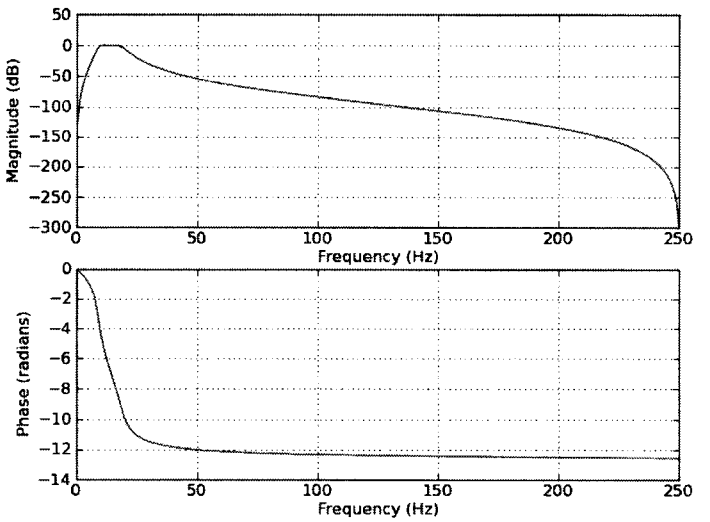

Figure 6.10: Butterworth Magnitude (dB) and Phase Response

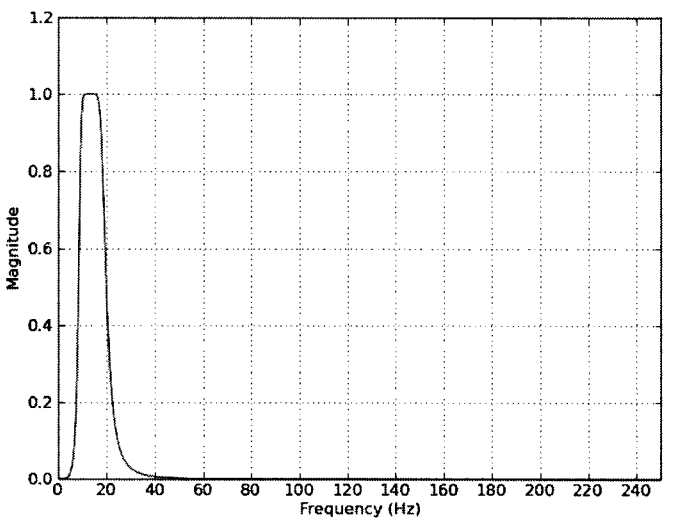

Figure 6.11: Butterworth Magnitude Response

desired results of a maximally flat magnitude passband and a linear phase response in the passband.

6.3 Design of the Desktop Software

Alongside the DAU and sensors is a version of the software intended for use on a typical desktop/laptop Windows or Mac computer. The purpose of this software is to process the data files created during a VA session with the DAU, and use the greater CPU and display capabilities to provide more information than is available on the DAU.

Particularly, this includes showing more simultaneous information on the main screen, as show in figure 6.12, computationally expensive calculations like the phase and eccentricity, and the report generation feature.

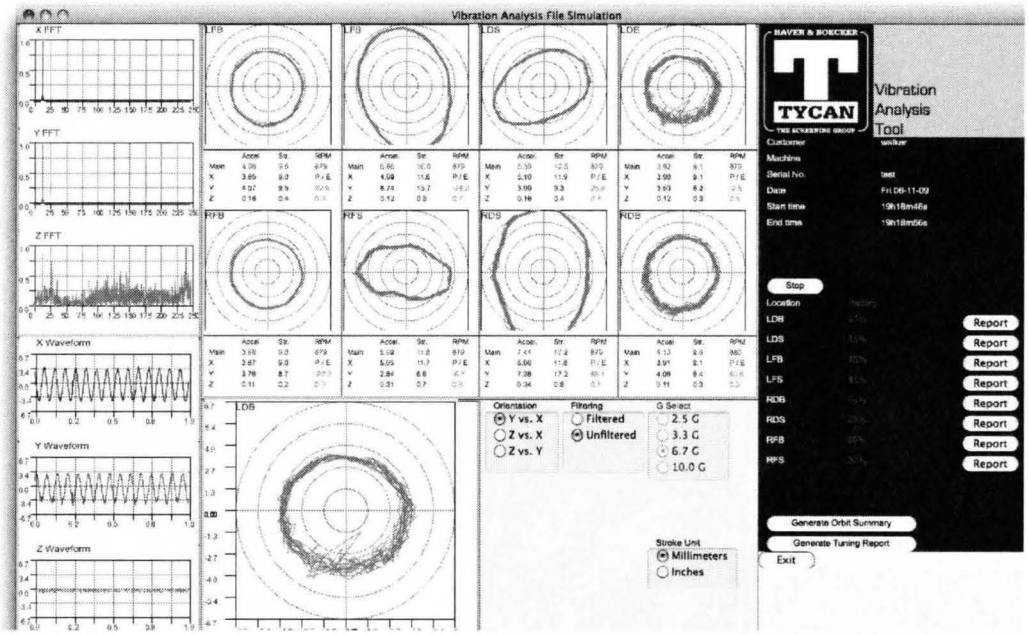

Figure 6.12: Main PC View

6.3.1 PCViewer

In short, the PCViewer module performs the same duties as the PDAViewer module, except with the goal of rendering the display on a desktop/laptop PC. More data is available on screen when using the PC software, so more display components must be laid out, but in general the same data is available. On the DAU a technician cannot simultaneously look at the FFT, waveform and orbits for a sensor, but all can be seen at the same time on the PC thanks to the increased screen real-estate.

Two numerical values that are available on the PC but not at all on the DAU (because of computational expense) are the calculated phase and eccentricity for an orbit.

The phase of an orbit is defined as the angle of the major axis of the orbit from the X axis of the plot, while the eccentricity is a measure of how "circular" an orbit is. A value of 0 for the eccentricity indicates a circle, while a value of 1 indicates a straight line.

6.3.2 ReportGeneration

The report generation capabilities of the software are centred around creating PDF reports based on certain calculations often performed on accelerometer data. The work in this module is essentially a straight port of pre-existing (and confidential) Microsoft Excel spreadsheets provided by an industrial partner, so they will not be analyzed as part of this thesis.

6.4 Vibration Location Detection Tool

In Chapter 4, a technique for using cross-correlation as an ideal filter was introduced, along with a description of using this technique to aid in the localization of vibrations on a rotating machine.

The VA system described in this thesis was built in part for the purpose of employing this new technique. The requirement for multiple sensors operating simultaneously was driven by this technique, as well as other reasons listed previously.

A tool, the Vibration Location Detection Tool (VLDT) has been developed to perform automated vibration location detection. It is currently separate from the

main VA system, but uses the data files created by the system. Future versions of the DAU software could incorporate it directly.

The VLDT is *not* a real-time tool. It is a post-processing step, requiring the results from a full VA run before it can be used, much like the output of the ReportGeneration module.

6.4.1 Design

The basic structure of the VLDT is a system that reads data files created by the DAU, analyzing them to identify vibrations, and creating a plot for each vibration to illustrate that vibration's affect on the machine.

Technically, the VLDT can only currently be applied to vibrating screens, with data recorded by the DAU described above. However, the necessary components that are specific to the DAU and vibrating screens are modular, and can very easily be swapped out. Namely, the module for reading data files generated by the DAU could easily be replaced if a new source of recordings were to be used, and the module for creating the visualizations of the vibrations on the vibrating screen could easily be replaced with one that does visualizations for another structure.

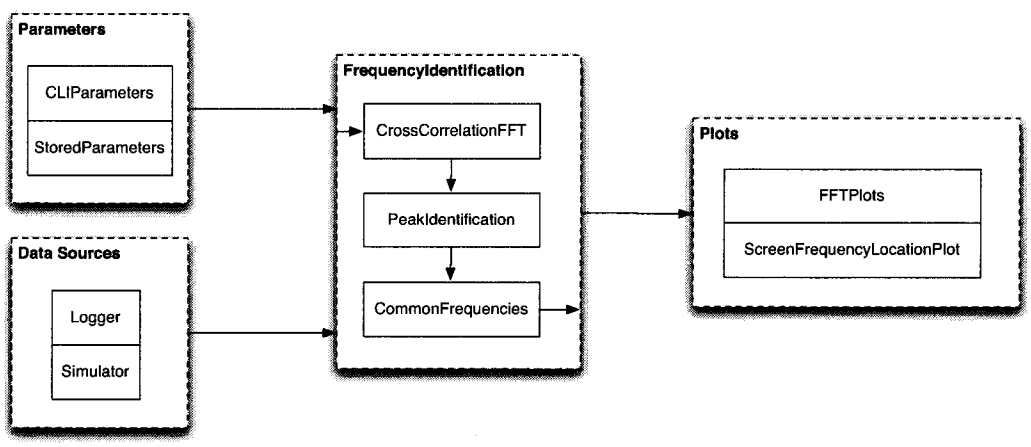

Figure 6.13: Vibration Location Detection Tool Components; Arrows show data flow

Figure 6.13 shows the structure of the VLDT. Two inputs are required up front: Parameters for frequency detection, and the data files (DataSources) generated by the DAU.

The connection between the DAU and this tool is the Logger module. This module is used by both software applications. It entirely encapsulates the process of writing to and reading from data files generated by the DAU. This module could easily be replaced in the future to enable reading from a different vibration analysis tool.

Alternatively, a Simulator module is present. This can be used to configure simulated vibrated screens, with vibrations present at different locations and with different amplitudes and frequencies, for the purposes of testing the VLDT's ability to detect those vibrations.

The FrequencyIdentification block of the diagram contains the three modules which perform the actual mathematics and computations of vibration location detection. CrossCorrelationFFT implements the techniques of section 4.6, performing the necessary cross-correlations and FFTs amongst all the sensor recordings. This data is passed to a PeakIdentification module, where frequencies from the FFTs are isolated and identified (this process is described in section 6.4.4. Finally, the CommonFrequencies module identifies vibration frequencies present in multiple locations on a machine, and packages them together.

The results of this block are passed to the Plots block, where appropriate visualizations are created, displaying the presence of vibrations through a screen. If a different vibrating machine were under analysis, the ScreenFrequencyLocationPlot would simply be replaced with a different visualization module.

To repeat, the only components that would need to be changed to support different types of vibration machinery are the appropriate module in the DataSources block and the ScreenFrequencyLocationPlot module.

6.4.2 Module Guide

Name	CLIParameters
Service	Responsible for reading/parsing command-line thresholds
Secret	Command-line argument parsing algorithms

Table 6.22: Module: CLIParameters

Name	StoredParameters
Service	Stores default command-line parameters
Secret	Data structures for storing and retrieving default parameters

Table 6.23: Module: StoredParameters

Name	Logger (Same as DAU Logger)
Service	Reads data samples from a DAU created log file
Secret	Data structures of log files

Table 6.24: Module: Logger (Same as DAU Logger)

Name	CrossCorrelationFFT
Service	Computes the FFT of the cross-correlation of two inputs
Secret	Algorithms for computation of FFT and cross-correlation

Table 6.25: Module: CrossCorrelationFFT

Name	PeakIdentification
Service	Identifies peaks in FFT to isolate frequencies
Secret	Algorithm for identifying peaks

Table 6.26: Module: PeakIdentification

Name	CommonFrequencies
Service	Determines frequencies common to multiple sensor points
Secret	Algorithm for determining common frequencies

Table 6.27: Module: CommonFrequencies

Name	FFTPlots
Service	Generates raw FFT plots
Secret	Algorithms for working with drawing libraries

Table 6.28: Module: FFTPlots

Name	ScreenFrequencyLocationPlot
Service	Creates vibration location visualizations
Secret	Algorithms for interpreting identified and rendering frequencies

Table 6.29: Module: ScreenFrequencyLocationPlot

6.4.3 Simulation

To illustrate the VLDT and the Simulator module, a simulation was created reflecting one possible fault situation. In particular, a screen is operating at 15Hz and two faults are present on the screen, one directly between points RFS and RDS, with another between LDB and LDS.

The data for each sensor is a combination of four possible data sources:

- The fundamental operating frequency, 15Hz, amplitude 3, random phase
- Fault 1, 80Hz, amplitudes from 0.1 to 2
- Fault 2, 115Hz, amplitudes from 0.3 to 3
- Gaussian random noise, zero-mean, standard deviation of 3

All sensors receive the fundamental frequency at the same amplitude, but the phase is randomly generated for each.

The amplitudes of Fault 1 and 2 are determined based on how far the sensor is from the fault source. Fault 1 is centred between RFS and RDS, and fault 2 is between LDB and LDS. Table 6.30 shows the particular contributions of each.

It is important to note that the amplitudes of the fault frequencies are all small compared to the amplitude of the Gaussian noise. While a normal FFT is usually fairly good at pulling out frequencies from noise, figure 6.14 shows the FFT plots for sensors LDS and LFS. Both of those sensors are receiving the 80Hz fault signal

Sensor Location	Fault 1 Amplitude (80Hz)	Fault 2 Amplitude (115Hz)
LDB	0	3
LFS	0.1	1
LFB	0	0.5
LDS	0.1	2
RDB	0.8	1
RFS	2	0
RFB	0.8	0
RDS	2	0.3

Table 6.30: Cross-Correlation Simulation Data Sources

at an amplitude of 0.1, and this is not strong enough to come through to the FFT. Instead the FFT only shows the fundamental frequency at 15Hz and the 115Hz fault frequency.

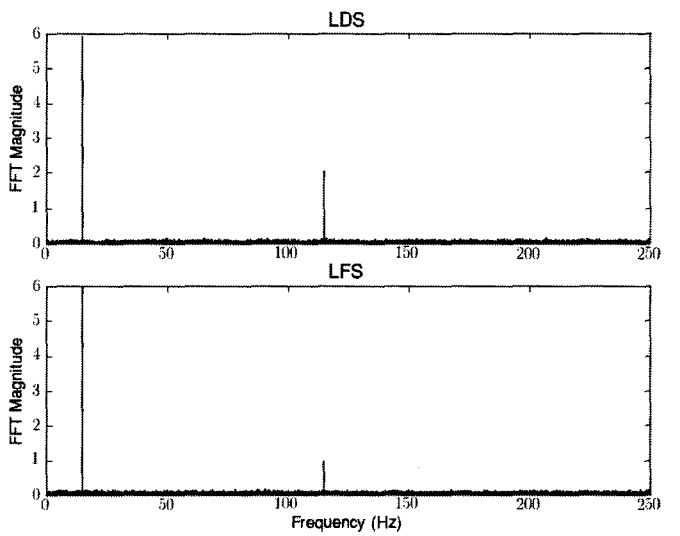

Figure 6.14: Buried Fault Signal; Only the $115Hz$ fault signal comes through the FFT, the $80Hz$ signal is too small and gets buried in noise

A technician looking at these FFT plots for LDS and LFS would assume that

the fault frequency is not being transmitted as far as those sensors, and any filtered-orbit analysis would eliminate those frequencies anyway, so the orbit would offer no additional information.

Looking at all the FFT plots, the technician would notice that other frequencies are present. For instance, figure 6.15 clearly shows two fault frequencies present in addition to the fundamental frequency. However it is difficult to localize where the various fault might be originating from. This is the power of cross-correlation and in particular, the power of the tool we have developed.

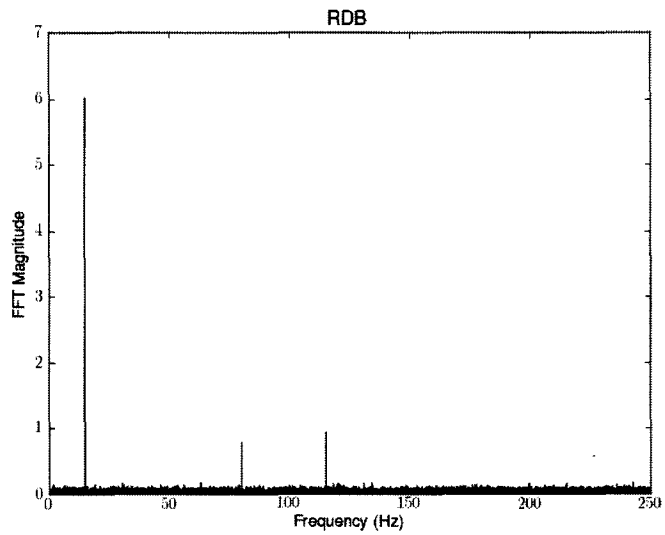

Figure 6.15: Fourier Transform of RDS signal

As mentioned, the tool performs a cross-correlation between each possible pair of sensors, and performs an FFT on this result. For each resulting FFT, our custom peak-finding algorithm (section 6.4.4) is employed to automatically detect frequencies that are present in the FFT.

Figure 4.2 shows the result of this for the fault frequency at 80Hz, and figure 6.16 shows the result for 115Hz. Recall that the simulated data centred the 80Hz fault between the RDS and RFS sensors, and the 115Hz fault between LDB and LDS. The plots clearly show to the technician where the fault signals are most strongly correlated, data that was never automatically available before.

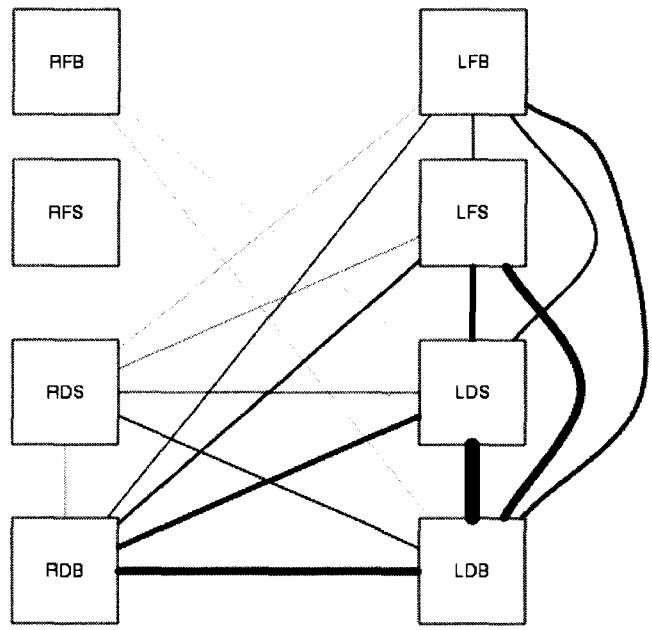

Figure 6.16: Fault Location Detection at 115Hz; Fault is centred between sensors LDB and LDS

6.4.4 Peak-Finding

A common task surrounding FFT processing is identifying the fundamental frequency in the analyzed signal. Typically this is done with the *argmax* function, simply finding the point in the FFT with the highest magnitude. This was shown in section 2.7 as a reasonably simple task.

For the purposes of the fault localization tool developed here, this is not enough. Instead of identifying just the strongest peak in the FFT of, the goal instead is to identify *all* peaks. While it could be left up to the user of the tool to manually identify all the peaks resulting from the FFT, the entire goal of the tool is to *automatically* do the work of fault localization. As such, user intervention is not desired.

The basis of the peak detection algorithm is slope changes, which is often the case in local maxima/minima algorithms. The basic algorithm is based on the work in [57] but with modifications.

The algorithm iterates through all the points of the FFT. As soon as the slope changes from negative to positive, or two positive slopes in a row are seen, then

the variable `can_look_for_peak` gets set to `True`. These two cases are shown in figures 6.17 and 6.18, respectively.

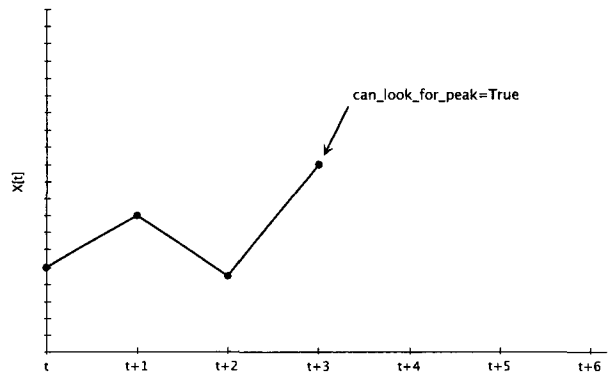

Figure 6.17: Negative-to-Positive Slope Change

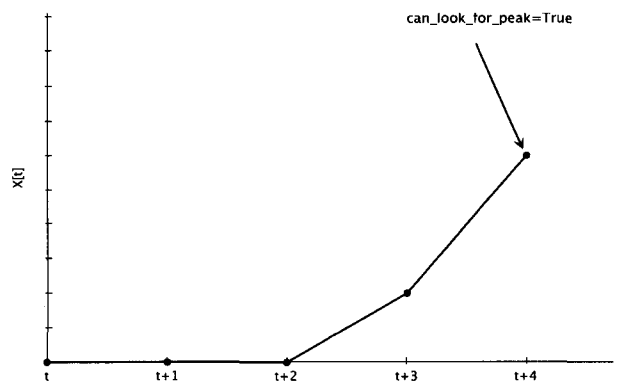

Figure 6.18: Positive-to-Positive Slope

The high-level steps the algorithm takes are:

1. Check for a negative-to-positive slope change or consecutive positive slopes
2. Set `can_look_for_peak` to True when either occur
3. Begin looking for candidate peaks
4. Mark a peak at point t as a candidate if its value $X[t]$ is greater than α (figure 6.19)

5. Check if $\forall(0 \leq i \leq \tau | X[t] > \Delta * X[t+i])$

6. If that condition is met, then mark point t as a peak, and continue

where

α A minimum threshold that a peak must pass to be considered

τ The number of points after point t that must be less than $X[t]$ to keep t in consideration as a candidate peak

Δ Used to verify that the τ points after t are all less than $X[t]$ by a certain value

These three parameters are all configurable in the tool, but empirical testing has found that default values of

$$\alpha = 5.0$$
$$\tau = 4$$
$$\Delta = 0.2$$

tend to produce very reasonable results.

Marking a peak as a candidate is shown in figure 6.19. The plot had a negative slope from $t+1$ to $t+2$ and a positive slope from $t+2$ to $t+3$, allowing the algorithm to check the next peak $t+3$ for possible candidacy. $X[t+3]$ is greater than the chosen α, so $t+3$ is marked as a candidate peak point.

When the condition in figure 6.19 is met, a special variable `peak_found` gets set to `True`, at which point the algorithm inspects the next τ points after $t+3$. Each of these τ points must have a magnitude less than a factor of $\Delta \cdot X[t+3]$. Figure 6.20 shows a case where this restriction fails immediately on $t+4$. In this case the point $t+3$ has its candidacy removed, `peak_found` is reset to `False`, and the algorithm returns to the basic slope checking. Figure 6.21 shows the condition passing for all τ points.

6.5 Description of the Data Acquisition Hardware

For completeness, aspects of the hardware platform used for the DAU are presented. These are specific based on choices and needs of a VA system for vibrating screens.

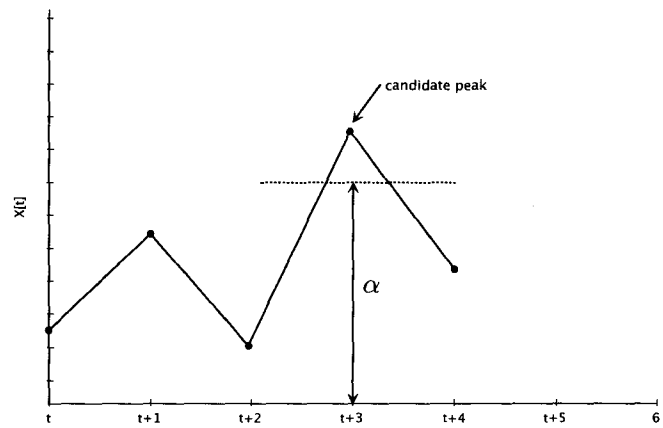

Figure 6.19: Marking a Peak as a Candidate

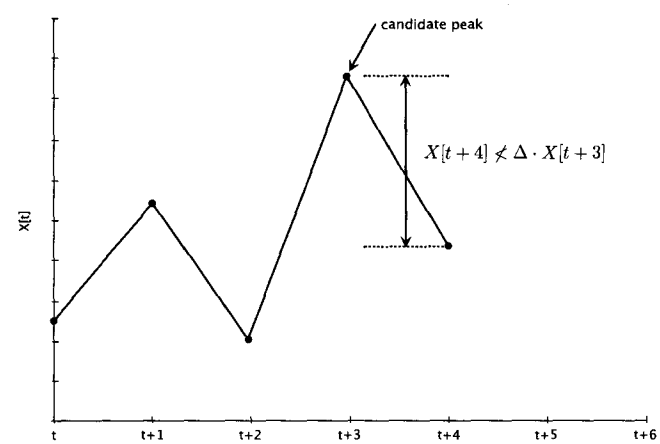

Figure 6.20: Peak Not Passing the Δ Criteria

While the sensors themselves were custom-designed for this research, as nothing appropriate was available "off-the-shelf", the hardware components of the Data Acquisition unit are developed by commercial companies. Specifically the main unit (the Personal Digital Assistant, or PDA) is a Tripod Data Systems (TDS) Nomad outdoor rugged handheld computer [58] running a version of Linux provided by SDG Systems Inc.

While this unit contains an internal Bluetooth transceiver, a USB-based external Bluetooth dongle was also selected for use with the system.

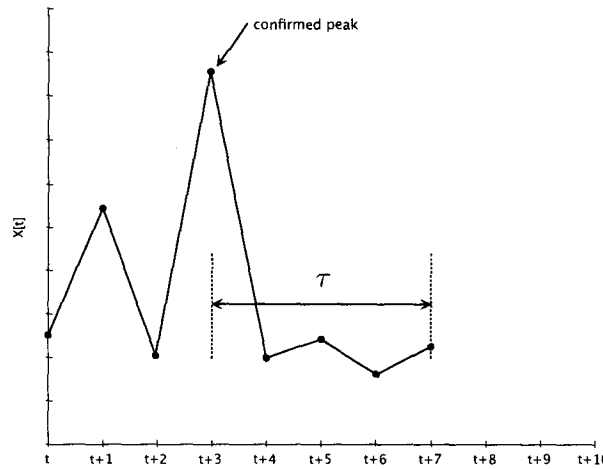

Figure 6.21: Confirming a Peak

6.5.1 PDA

The brochure for the Nomad states [58]

> ... the Nomad computer meets rigorous MIL-STD-810F military standards for drops, vibration, humidity, altitude and extreme temperatures. It also comes with an IP67 rating. That means the Nomad handheld is completely sealed against dust, and it can survive immersion in up to a meter of water for 30 minutes.

Being sealed for dust and water, as well as high standards for drop resistance were essential to the choice of this unit. The Nomad has an 806MHz ARM [59] processor, 128MB or RAM and 512MB of Flash storage. These were found to be adequate for the computational requirements of the DAU. The display is a 480x640 VGA TFT touchscreen LCD. Many other handheld units that were found had resolutions much lower than this, and because of the requirement to collect data from eight sensors simultaneously, the highest resolution possible was needed to show data for all eight sensors at the same time.

The Nomad also incorporates an internal Bluetooth transceiver, an SD-card slot, a CompactFlash card slot and full USB2.0 host *and* client ports.

6.5.2 USB-Based Bluetooth Transceiver

While the Nomad contains an internal Bluetooth transceiver it is unfortunately inadequate for the needs of the system. Specifically it is only a Class 2 Bluetooth transceiver, and as mentioned in section 5.3.3 this provides only a very short theoretical maximum operating range. Even if it were a Class 1 device, there would still be the issue from section 5.1 wherein a single Bluetooth network can only contain eight devices. With the Nomad being one required device in the network, this would only allow communication with seven sensors.

To solve both of these issues an external USB-based Bluetooth dongle is attached to the Nomad during operation. This is a Linksys USBBT100 Class 1 unit [60], shown in figure 6.22. This was chosen over a multitude of other Class 1 USB devices because empirical testing showed it had the best range. Unfortunately most manufacturers of USB-based Bluetooth devices do *not* list the *dBm* rating of the device, only if it is Class 1, 2 or 3. Linksys does give a rating for this device, at $13 - 17dBm$. Multiple units had to be purchased and tested before the Linksys device was decided on.

Figure 6.22: Linksys Bluetooth USB Adapter USBBT100

Not only does using this dongle allow the system to operate at Class 1 ranges, it also provides a way for skirting around the eight device limit. While the Linksys device does not allow the unit to communicate with anymore sensors than any other Bluetooth device would, what it does do is provide a second Bluetooth device for the Nomad.

When seven or fewer sensors are going to be used for data acquisition, the software only makes use of the Linksys device. When eight sensors are desired, the software has the Linksys communicate with seven of them, while using the internal Bluetooth for the eighth. So in essence two separate Bluetooth networks are running simultaneously.

There is still the problem that the internal Bluetooth transceiver of the Nomad

is only Class 2, so the software allows the user to specify which sensor they are physically closest to, and the software will have the internal Bluetooth communicate with that sensor.

6.5.3 Linux Operating System

While not actually a piece of hardware, it is important to note that the Nomads purchased for the DAU all come loaded with a customized version of the Ångström OpenEmbedded Linux Distribution [61]. This was actually a major factor in the selection of the Nomad, as the Linux distribution is essentially equivalent to what one would find on a desktop or server Linux machine, meaning the majority of open-source software for Linux would simply work, and not require any modifications for running on a handheld device.

This is in contrast to the Windows Mobile-based devices that were found, where only software specifically made for Windows Mobile (which is distinct from the standard Windows operating systems) would work.

In particular, using Linux allowed the software to make use of the standard GTK [62] and wxWidgets [63] window and widgets systems, the NumPy and SciPy numerical computation libraries and recent versions of the Python [64] programming language.

6.6 Cython

Certain components in the software were not implemented in Python, the main language used, but instead in a related language called Cython [49].

A common technique in Python programming is to write C-based modules (called "extension modules") where performance is vital. Being an interpreted language, Python is often not as fast as compiled C-based software. A C-based API is available for writing modules in C that can be imported and used in a Python program. This allows performance critical components to be written in C and used from within Python.

Cython is a speciality language that itself compiles down to C modules, which can then be compiled such that they are directly usable in Python. The advantage of using Cython over programming directly in C is that the Cython language is itself *very* similar to Python, and the programmer does not need any knowledge of the C-based API required by Python. Thus the amount of work required to

implement a component in high-performance C is greatly reduced from working directly with C.

In simple terms, Cython is Python with explicit static-type declarations. A number of base types (with direct mappings into C) are defined in Cython, such as integers, floats, character arrays, etc., and variables and functions can be marked as being of that type at compile time. This allows for compile-time optimizations not available with the dynamic run-time type system used in Python. This provides impressive speed gains for certain classes of problem.

Every module in the software was first written in Python, and only transitioned to Cython when performance was found to be lacking. In the end, every single module in the "Collect and Process" group was eventually transitioned to Cython. These modules are all used extensively (each is invoked in almost every loop iteration of the Reader module's main loop), and many are computationally expensive. The DAU itself has limited computational resources (compared to a modern desktop or laptop computer), so these speed gains were necessary.

In the "Display" group, only the Scalers and CircularQueue modules were moved to Cython. They contain functions that are called almost continuously, and impressive performance increases were gained with the move to Cython.

Chapter 7

Analysis of Wireless Network Issues

To reliably use this system not only for the capabilities of the current software, but as well for future versions of the software, a catalogue of existing and potential system issues must be examined and analyzed.

One problem with the system is currently known, and will be discussed at length in this chapter. Namely the issue of synchronization delays between the nodes in an active sensor network.

This problem is not unique to wireless VA systems, but any system with wireless sensors that require a strong degree of synchronization. The use of Bluetooth added an additional layer to the issue, one that any researcher using Bluetooth for wireless networks should be made aware of.

7.1 Description of Synchronization Problem

In addition to the cross-correlation analysis described in Chapter 4, some work was started on using lag calculations from cross-correlations to determine phase differences between physical locations.

To begin this work with the VA system, the synchronization level of the sensors needed to be tested. The first tests were performed by placing eight sensors on a flat, stable surface. Recording was started, and impulses were introduced by hitting the surface with a solid object. These tests were called "Spike Tests", and were performed to see what kind of time lag, if any, existed between the sensors. For certain kinds of cross-correlation analysis (including the work of Chapter 4), the lag does not really matter, but for phase analysis, an understanding of lag within the system was vitally important.

The first results are shown in table 7.1. This shows the number of samples between a given pair of sensors recording the same event.

	LDB	LFS	LFB	LDS	RDB	RFS	RFB	RDS
LDB		-14	-38	-80	83	-65	-55	-26
LFS			-24	-65	97	-51	-40	-12
LFB				-41	100	-27	-16	13
LDS					95	14	25	54
RDB						-81	-95	-88
RFS							11	40
RFB								29

Table 7.1: Eight Sensor Spike Test; Sample lag between eight sensors

The NumPy function `xcorr` (similar to the MATLAB function of the same name) was used to perform cross-correlations between each of the sensors. It takes two data sets, and returns a list of lags (of length $2 \cdot maxlags + 1$), and *correlations*, a $2 \cdot maxlags + 1$ length correlation vector. The conditions for the recording consisted of five distinct spikes on the surface, so the lag for each cross-correlation giving the strongest correlation should represent the point in time when both sensors saw the spikes. For each pair of sensors the lagged cross-correlation was calculated as in 7.3.

$$(lags, correlations) = xcorr(x, y) \tag{7.1}$$

$$max\_correlation = \underset{x}{\text{argmax}}\ correlations \tag{7.2}$$

$$max\_lag = lags(max\_correlation) \tag{7.3}$$

The value *max_lag* shows how many samples apart two sensors were when viewing the same physical event. The worst result happened to be between sensors RDB and LFB, where there was a full 100-sample difference in the time between recordings. At 2 ms between samples, this equates to a full 200 ms difference between the time that RDB recorded a spike and LFB recorded the same spike. This is obviously unacceptable if trying to use the DAU to detect phase delays between physical events occurring in a vibrating system.

What is the source of this error? How could it be that two sensors recording the same physical event would report that event 200 ms apart?

Four separate issues revealed themselves:

1. Unsynchronized reception of "Start sending data" message
2. Lack of proper time stamp
3. Unsynchronized recording between SensorManager instances
4. Propagation time of data through communication stacks

Each of these issues can be traced back to improper synchronization of the wireless network. Synchronization of wireless sensor networks is actually a field with active research. A brief summary of the current research will be presented, before returning to the problem of these four issues .

7.2 Synchronized Wireless Sensor Networks

In a Wireless Sensor Network (WSN) where distributed nodes take sensor readings from their environment, time synchronization between the nodes is needed for two main purposes:

1. Scheduling when nodes should coordinate to simultaneously take samples
2. Time stamping the individual samples that a node records

In traditional networks connected via a physical medium, time synchronization is a well understood problem. These networks either have a dedicated cable used exclusively for propagating a clock signal, or use a data channel for synchronization, embedding the clock signal inside the data.

Recent advances in component miniaturization and low-cost/low-power device designs have increased not only research into distributed WSNs, but also practical implementations [65]. These networks must be highly energy efficient and cost effective to make their use practical in the real world. In response to this, new algorithms have been developed to facilitate clock synchronization in systems lacking a physical clock signal medium. Time synchronization is just as important in wireless sensor systems, but the problem is much more difficult.

One classical solution is to employ GPS receivers on each sensor node. GPS provides high accuracy, better than $200\mu s$ relative to UTC [66]. Unfortunately GPS receivers still tend to be expensive, both in cost and energy use, to make them a viable solution for low-cost high-density sensor networks. GPS also requires a line-of-site view of the sky, limiting them to outdoor scenarios.

More realistically, algorithms have been developed for propagating a clock signal through the same wireless medium that the data is transmitted.

The two main systems for clock synchronization are *Master-slave* and *Peer-to-peer*. In a master-slave protocol, one node or unit is defined as a master, and all other nodes are the slaves.

Many examples of protocols based around master-slave can be found in the literature. Arias et. al [67] built a sample WSN wherein a beacon broadcasts a synchronization message every T seconds to each node in a three-node network. The nodes are sampling a signal at 400Hz, and a general purpose output line on the node is measured with an oscilloscope to record when it actually performs its sampling. Using just their simple synchronization procedure, the authors were able to show that the nodes could be kept in sync if a maximum inter-node phase of 100μs is allowed.

Mock et. al. [68] were able to use the IEEE 802.11 clock synchronization procedure, which employees MAC level clock synchronization.

A particularly interesting mechanism is described in [69], entitled Reference-Broadcast Synchronization (RBS). In this system nodes send reference beacons to their neighbours using MAC or physical-layer broadcasts, so instead of coordinating with a central unit, receivers coordinate with *each other*. The beacon message contains no time stamp, nor does it need to.

An excellent survey of clock synchronization systems for WSNs can be found in[70].

Bluetooth wireless networks in particular present issues to network synchronization. One of the best papers found discussing this topic is [71]. The authors not only analyze the issues surrounding synchronization with Bluetooth, but also propose a new synchronization method, Broadcast Synchronization over Bluetooth (BSB), with included tests on their own implementation. They were able to find that the synchronization error average only 4.6μs, with a worst case error of 17.4μs. This compares to 16.9μs and 44μs for the TPSN method introduced in [72], and 29.13μs and 93μs for the RBS method.

Unfortunately BSB requires low-level access to the Bluetooth transceiver, on both the sensor-side and the DAU, that is not available to us with the selected hardware. It does prove though that Bluetooth can be a suitable choice for wireless networks requiring tight synchronization.

7.3 Analysis of the four synchronization problems

To reiterate, the four synchronization problems present in the system are:

1. Unsynchronized reception of "Start sending data" message
2. Lack of proper timestamp
3. Unsynchronized recording between SensorManager instances
4. Propagation time of data through communication stacks

7.3.1 Unsynchronized Reception of "Start sending data" Message

Problem 1 comes from the way in which the DAU tells the sensors to start transmitting their data. At the beginning of a session, after the DAU has connected to each of the sensors, the technician presses a "Run" button. This tells the DAU to send the "Start sending data" message (table 5.3) to each of the sensors. With eight sensors, this equates to sending eight separate messages, one to each sensor. Unlike 802.11 networks, Bluetooth does not *normally* allow for broadcast messages. Anything that must be sent to every sensor has to be sent individually to each.

This causes an immediate problem because each sensor will receive the message at a different time, and thus start transmitting data at separate times. Not only do each of the messages have to be transmitted separately, but there are multiple layers of software that each must pass through before actually passing through the air to the sensor, possibly changing the time it takes for each message to transmit.

At a very basic level, sending a single byte causes that byte to travel through an application layer, top-level kernel layer, link layer (Bluetooth driver) and hardware layer (Bluetooth transceiver).

Each of the layers of the stack potentially has its own buffers, and as the DAU is a full Linux system, we cannot predict when process switching might take place, meaning passing the same message through the stack multiple times might require a different amount of elapsed time.

The BSB synchronization method described in [72] proposes a solution to this problem. Some Bluetooth devices provide very low-level access to the HCI layer,

a base layer of the Bluetooth protocol used for implementing most of the higher-level functionality. HCI not only allows for broadcast messages (a single transmitted message simultaneously received by all listening devices), but presents an interface for detecting exactly when a message has arrived, before any processing of the message occurs. This would completely eliminate the problem of having to send multiple start messages, each having to individually pass through all the layers and buffers of a normal Bluetooth message. Figure 7.1 clearly shows the timing characteristics of using HCI for the BSB protocol. The physical flight time of the message TOF_i is negligible, and the reception times T_{BT_i} and T_{UC_i} will be essentially identical as long as the exact same Bluetooth devices are used.

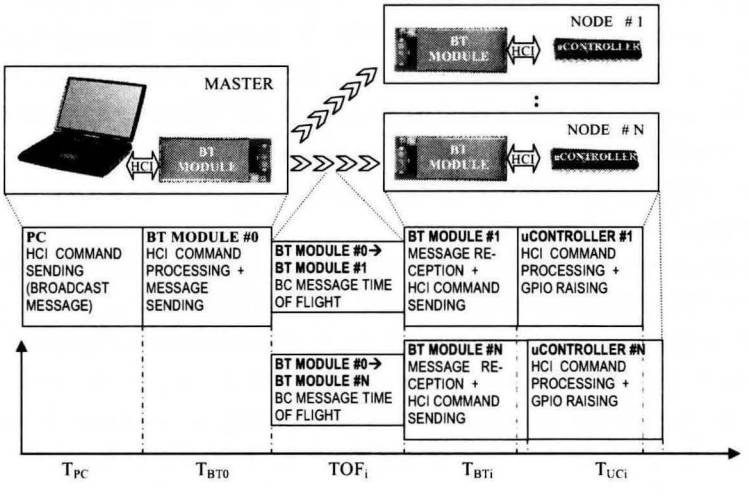

Figure 7.1: BSB Timing (image from [72])

Unfortunately the hardware selected for the sensors is incapable of using the HCI layer, so none of the currently manufactured sensors will ever be able to take advantage of this technique, and future sensors would require both a hardware and software redesign. Suitable Bluetooth hardware with enabled HCI *and* supported our power requirements could not currently be found anywhere.

This changes the timing diagram to require i T_{PC} times and i T_{BT0} time periods, each one being different.

Timing experiments on our system have shown that *typically* the total time between starting to send the "Start" message and reception of all messages, for eight sensors, is roughly 10 ms. Unfortunately every few tests this would jump to

as much as 40ms. If the time was consistently 10msthen post-processing could be done to improve the synchronization results, but with the occasional jump to 40ms, and no way to detect this 40ms jump, this cannot be done.

Even when the time does not jump to 40ms, the time between sensors receiving the messages varied drastically.

The timing experiments were performed by connecting eight sensors to a 16-channel digital oscilloscope, and setting a GPIO line high on each sensor when it received its individual start message. This let us compare the relative times that each sensor received the message.

Eight-sensor tests were performed, as well as five-sensor. The results of two of the five-sensor tests are shown in figure 7.2. Time $t = 0$ is interpreted as the time that the first sensor received its start message. The time differentials are then shown for every subsequent sensor receiving its message. Though both tests were performed under identical conditions, using the exact same sensors and DAU, the total time for all messages was almost double in the first experiment compared to the second, and the inter-sensor times varied quite a bit.

Multiple experiments were performed, but only the first two are shown, as they are fairly indicative of the rest.

The results of these experiments reflect just how variable the time required can be for the DAU to go from calling the appropriate `send()` function in code to the contents of that message actually being transmitted over the air. The software on the sensors is architected in such a way that the time between receiving a Bluetooth message at the transceiver and processing its contents is negligible, so the times shown in figure 7.2 are almost entirely from the DAU side.

7.3.2 Lack of Proper Timestamp

The next issue we discovered preventing proper wireless synchronization is the lack of a true timestamp on readings coming in from the sensors. Almost all of the published synchronization schemes begin with the master sending out some form of timestamp, so that all the sensors in a network have a common time point to synchronize on. When each sensor then sends a message with data, it attaches its own local time to that message (where the local time was synchronized with the master), letting the master know exactly when that particular data was recorded.

Our data messages from the sensors contain no such timestamp, as shown in figure 5.6.

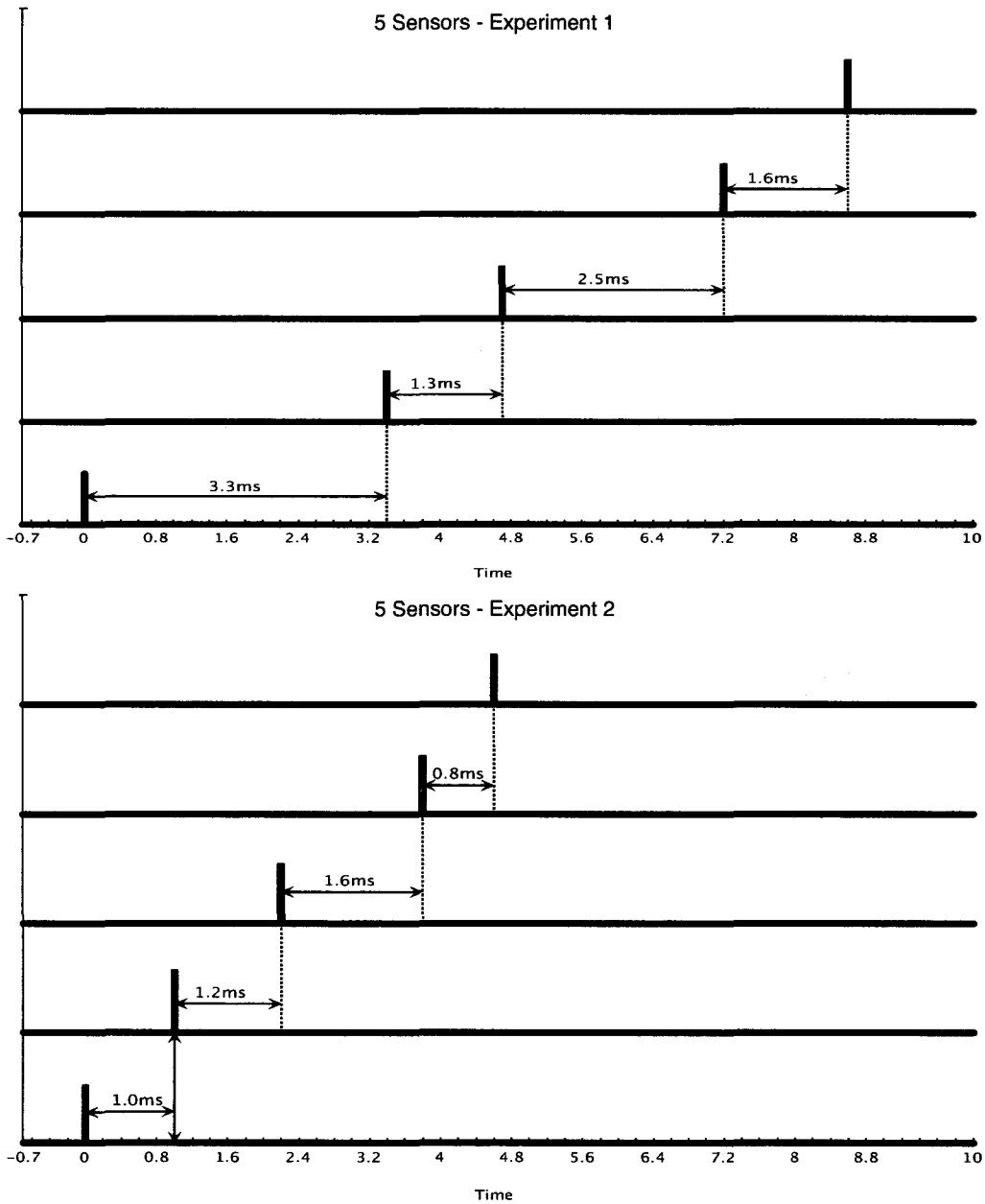

Figure 7.2: Timing Diagram

7.3.3 Unsynchronized Recording Between SensorManager Instances

A problem that arose from the implementation of data recording on the DAU was that the individual SensorManager instances were initially told to start recording in a fairly haphazard manner.

The SensorManager module is where data coming in from the sensors is actually recorded to disk. Each sensor connected to the DAU has its own SensorManager instance, and the (up to eight) SensorManager instances record the data coming into them into their own individual data files.

When a user of the DAU wishes to start recording the data that is coming in, they simply press the "Record" button on screen, and a message is sent to each of the SensorManagers telling them to start recording all of the data they process.

Unfortunately, no care was taken to look at the current progress of each of the SensorManagers. The DAU is a single processor system, so even with multithreading, no more than one SensorManager is actually processing data at any individual point in time. Because of this, the SensorManager for sensor i might be a few hundred samples ahead of sensor j when the "Record" message comes in. Those samples already processed for i will not be recorded, but the samples representing the same set of time for sensor j will be recorded. This will cause a huge skew in the final recorded data, because the first recorded sample for sensor i will not represent the same point in time as the first recorded sample for sensor j.

This has been verified as the prime contributor to the huge skew shown in table 7.1.

7.3.4 Propagation Time of Data Through Communication Stacks

The final issue is very much related to one of the problems preventing synchronization of the "Start sending data" message described in section 7.3.1. In that case, the multiple buffers caused variable delays in the transmission of the message, but all of the same communication stacks will have receive-buffers, causing variable delays in the reception of data from the sensors. When the `select()` function is used to query for the availability of new data, the Linux kernel may or may not return all currently available data. Just because the function returns data for only $m < n$ sensors does not necessarily mean that data is not already present for all n sensors.

This is one of the main issues causing the SensorManagers to process data

representing the same instant in time, at different times. A theoretical scenario might be:

1. Sensors i and j simultaneously transmit data representing time t, $i(t)$ and $j(t)$ respectively
2. `select()` is called in the Reader module
3. The Linux kernel has so far only processed data $i(t)$, so only sensor i is marked as currently having data by `select()`
4. SocketRead is informed that sensor i has data, but sensor j has none, so it reads the data in
5. Data $i(t)$ is passed to the SensorManager instance for sensor i, SensorManager$_i$
6. SensorManager$_i$ performs the DC and Butterworth filtering on the data, but it is not recorded
7. The user presses the "Record" button, which informs all SensorManager instances to start recording
8. The Linux kernel finally processes data $j(t)$
9. `select()` is called again, and sensor j is marked as having data
10. Data $j(t)$ is finally read by SocketRead and passed to SensorManager$_j$
11. SensorManager$_j$ DC and Butterworth filters data $j(t)$ and then records it

This clearly illustrates a very likely scenario in which data $i(t)$ and $j(t)$, representing the same moment in time do not both get recorded to the respective files for sensors i and j.

7.3.5 Summary

While four separate issues were detailed, in reality they are tightly interconnected, with one directly affecting the other.

The unsynchronized "Start" messages make it impossible for the sensors to have a common time reference. The lack of an explicit timestamp marker in the data packets is a direct result of not considering the common time reference problem. The unsynchronized recording of the SensorManager instances is partially

the fault of not being able to coordinate packets from different sensors, which is directly compounded by the various receive buffers in the different communication stacks causing multiple sets of data to arrive at the application level at different times, even if they were physically received over the air simultaneously.

7.4 Potential improvements to wireless synchronization

While a proper solution to the synchronization issue is essentially impossible without changing the Bluetooth transceiver on the sensors, various steps can and have been taken to mitigate the issue as much as possible.

7.4.1 Optimized `send()` Routine

Until recently, the software was sending the start character to each of the sensors in a fairly inefficient way. The software was looping through a list of Handler instances, building the send message individually for each, sending the message through the Handler interface, and iterating to the next Handler to perform the same step. This caused all of the time required to iterate through the list of Handlers, as well as accessing the Handlers interfaces, to be added to the time between actually calling the `send()` function for each sensor.

A very low-level optimization, requiring knowledge of the internals of the Handler module, was necessary to solve this.

While iterating through the list of Handler instances, a new list was created, containing function references to the actual `send()` function for each of the low-level Bluetooth sockets contained within each of the Handler instances.

Each sensor receives an identical start message, so the message was then constructed just once, and finally the list of function references was iterated over, calling each of the references with the same constructed message.

Now the only computation performed between each of the `send()` calls is the code necessary for iterating through the list.

7.4.2 Ad-hoc Timestamping

With regards to the lack of timestamp, figure 5.6 showed the structure of the data coming in from the sensors, and there is no explicit timestamp being used.

Fortunately though, there *is* the CNT byte appended to the end of the packet. As mentioned in section 5.2.10, this was put in place as a simple CRC. Because of the nature of how it's incremented, it *could* be used as an ad-hoc timestamp mechanism.

The CNT is known to always start at 0, and roll over after 255 (being a one-byte value). Because it is only incremented once for every 10 sample periods, and each sample period is 2 ms long, this one byte counter can be used to timestamp $10 \cdot 2\text{ms} \cdot 255 = 5120\text{ms}$ worth of time. Though it rolls over to 0 after 5120 ms, the DAU can simply keep track of the number of times it has rolled over, effectively implementing a full timestamp.

The main problem with this approach is that we still cannot guarantee at what point in time each of the original $\texttt{CNT} = 0$ messages actually come from. Even though experiments on the system showed that it typically takes 10 ms for all the sensors to receive their "Start" message, the inter-sensor receive times varied so much that we cannot really be sure when a sensor actually received it, relative to the other sensors.

7.4.3 Synchronized Recording in SensorManagers

As described in section 7.3.3, the SensorManager instances were not starting their recording in a synchronized way. The goal should be that the first sensor readings actually recorded by each SensorManager should all represent the exact same moment in time.

Because we have no way of actually synchronizing the incoming data with respect to each other, this cannot be fully accomplished. However the effects of it can certainly be mitigated. As was shown in table 7.1, the effects were fairly serious.

The easiest way to reduce the severity would be to track a little more state with respect to how many packets each SensorManager has processed. The example in section 7.3.4 showed how easy it was for the start point of the data files for two different SensorManagers could vary so greatly. If the technique for ad-hoc timestamping from the previous section were implemented, it would suddenly be very easy for each SensorManager to track exactly how many milliseconds worth of data they have already processed.

Let the variable $num\_packets_i$ represent the number of packets each SensorManager_i has processed, for $1 \leq i \leq n$ and n sensors.

When it is time to begin recording incoming packets, the software would simply find $max(num\_packets_i)$. Each SensorManager_i would then have to wait until

it had processed that many packets before it begins recording subsequent values to its file.

If all of the sensors were perfectly synchronized with respect to time, then this would completely solve the issue of delays in processing causing unsynchronized files.

Since the sensors are not synchronized, this will at best reduce the magnitude of the problem.

7.4.4 Replacement Bluetooth Transceivers

To attain true synchronization, the hardware on the sensors has to be changed, replacing the current Bluetooth transceiver with one that allows access to the HCI level of the Bluetooth communication stack. The changes suggested here will improve the situation to a degree.

The difficulty is finding a Bluetooth transceiver that provides HCI access *and* is powerful enough to meet the environment requirements of vibrating screens. Thus far, such a device has not been found.

7.4.5 Cross-Correlation Lag Adjustment

The cross-correlation technique used in section 7.1 to detect the lag error between two sensors can also be used to perform posterior synchronization on the recorded data files.

If the cross-correlation between two sensors shows that the maximal correlation comes at a lag of m packets, then the first $m-1$ packets can simply be subtracted from the second data file.

The signal providing the greatest amplitude for the sensors will be that resulting from the main operating frequency of the machine, so the point of maximum correlation should occur when that component of each signal overlaps in the correlation.

7.5 Practical Consequences

While a great detail of discussion has taken place regarding the synchronization issue, a simple fact is that this problem does *not* affect VA as is currently being performed, nor does it affect much of the advanced analysis proposed in this thesis.

Quite simply, the lack of microsecond synchronization really only affects the ability to analyze the cross-correlation of sensors for lag differences.

Each sensor is individually recorded correctly, so individual comparisons, which make up the bulk of current VA techniques, are not affected.

The lack of synchronization does not affect the ability to perform frequency domain analysis (phase response comparisons not withstanding).

It does not even affect the proposed cross-correlation techniques from section 4.1. The point of that work is to look at the resulting frequency peaks after an FFT is performed on a cross-correlation result. Even if the phase between two sensors is measured incorrectly, the periodicity of the cross-correlation remains the same.

For sensors in close proximity to each other on the vibrating screen, phase differences are already unlikely (when assuming the screen as a rigid body, at least within small sections), so the techniques from section 7.4.5 should be fairly reliable.

Using this technique, posterior synchronization of the recorded data files can be performed. The main operating frequencies will definitely be synchronized, and producing the main peaks of the cross-correlation. It is then not unreasonable to do phase-based analysis of any possible fault frequencies in the system.

Chapter 8

Conclusion

Signal analysis is an incredibly important topic, relevant to hundreds if not thousands of fields. One such field, Vibration Analysis – which itself can be broken into many sub-fields – was the topic of focus for this thesis. Vibration Analysis makes use of many of the traditional signal analysis techniques, time-domain and frequency-domain, while also presenting certain challenges with regards to signal capture.

Depending on the particular vibrating structure under analysis, conditions can be quite extreme. Some structures are massive in size, while others reside in hazardous environments. Vibrating screens, the particular type of system used in this thesis, are often present in mines or mounted atop multi-story open-air structures. Safety is a constant concern for VA technicians, and the tools which they use to perform VA must be both convenient to operate in these environments, as well as reliable. A reliable system plays a major role in reducing the amount of time required to actually perform the analysis, thus reducing the amount of time necessary in the environment.

Signal analysis for VA and the development of tools for performing the signal capture and analysis were the two main focuses of this thesis. We presented a new technique using cross-correlation for identifying the levels and locations of harmonic components present over a rotating machine, as well as designed and developed a new VA system capable of performing the necessary signal capture and analysis under the often extreme conditions faced by VA technicians. This VA system is now professionally manufactured, commercially available, and in use by dozens of Vibrating Screen technicians around the world.

The cross-correlation technique performs two tasks:

- Utilizes information from multiple sensors to eliminate noise and identify

frequencies

- Provides a physical mapping of frequency content to physical machine locations

Many of the traditional filtering techniques employed in VA require the technician to already know what frequencies to look for, or are actually more appropriate for machine tuning and maintenance. The bandpass filters often used must be placed around some frequency component. If there are known possible fault frequencies on a machine, then the bandpass filter can be centred there and the output analyzed. Often though the filter is centred around the operating frequency of the machine, which helps emphasize how well the system is performing within its designed parameters, but has the negative side-effect of eliminating all other frequency information.

The cross-correlation technique makes no assumptions about which frequencies should be analyzed. Instead it uses cross-correlation between simultaneous sensor readings to automatically eliminate noise and identify which frequencies are present. By utilizing the data from multiple sensors, the cross-correlation technique is often able to identify the presence of frequencies that the traditional FFT would miss. While it cannot be strictly said that the presence of a frequency indicates a fault is occurring, it is the case often enough that any detected frequency outside of the operating frequency warrants investigation by the technician. By first identifying frequencies, and then providing a physical mapping of how they are occurring on a machine, this technique provides a powerful new tool to VA technicians.

In short, filtering as has been traditionally performed is not necessarily the right way to approach fault detection. Too often important frequencies are filtered out, or too buried in noise to be detected, and cross-correlation opens the door to interesting possibilities for fault detection.

The VA system we designed was created not only to provide improved hardware and software compared to what technicians were currently using, but also to take advantage of this cross-correlation technique. With its wireless sensors it more easily enables multiple simultaneous sensor recordings, a task that would be onerous with a portable wired system. In addition, the system allows technicians to record vibration readings in a much safer manner. Rather than being physically tethered to a machine which rotates at over 6G and processes hundreds of tonnes of material per hour, they can instead place a network of sensors over the machine, then stand at a safe distance to record the measurements and analyze the output.

The VA system has been designed in a very general and modifiable way, and the overall design and implementation has been presented with more specificity than we have seen in the literature. Our hope was that such a development would reduce the implementation complexity for future researchers hoping to create similar systems. In fact, work has already begun to take the developed system and modify it for use in a fixed-installation continuous monitoring setup for Vibrating Screens.

Along with presenting the design of the system, we have also discussed a few issues that presented themselves during the development, which anyone designing a similar system should be interested in. Of these, two in particular stood out. The first was related to the amount of raw data we were attempting to transmit over a Bluetooth network. This presented problems we did not foresee at the outset of the project. It caused issues not only in how low-level communications were handled between components on the sensor itself, but also in the final sampling rate we could use for the system. Second was an issue related to the ability to perform correct time synchronization between the sensors. Without very specific hardware, it is simply not possible to use Bluetooth and have a level of synchronization between the sensors that would be necessary for phase-related cross-correlation analysis. While this did not affect the work presented here, anyone who wishes to use the system as-is for phase analysis must be wary. Future versions of the sensors should be able to easily correct this problem.

Knowing what we know now about the wireless system, it might be worth considering 802.11 WiFi for future systems. While the power drain is *much* higher than Bluetooth, data transfer speeds are also much higher and time synchronization is much easier to perform.

As the costs for sensor systems continues to drop, and the availability of high-precision sensors rise, it should be expected that more and more structures that require VA will begin to ship with the necessary sensors built into the structure itself. Be this as it may, the need for portable VA systems will be present for decades to come. Too many systems which require VA are already out in the field, and most of these will not be retrofitted with permanently installed sensors. Our hope is that the system developed here and the lessons learnt can provide guidance to those building future generations of wireless VA systems.

8.1 Future Work

Regarding the cross-correlation technique, the next steps should be to attempt to use it for feature extraction within some existing expert systems. All of the expert systems discovered in the research based their feature extraction on a single sensor, so a great deal of modification to these systems might be required to enable this.

Work has already begun on modifying the VA system presented here for the purpose of a permanently installed, continuously monitored VA system for Vibrating Screens. However more work could be done on the current system, to both improve the wireless synchronization of the sensors, and to add additional forms of VA. Some of the wavelet-based noise elimination techniques could be employed, to identify frequency content only present at one physical location (which would make it undetectable using our cross-correlation technique).

We had planned on implementing a wavelet analysis system to enable time-frequency analysis of the screens, but this was eliminated from this stage of development. For typical issues with Vibrating Screens, the frequency characteristics will not change over a short period of time, so this type of analysis probably would not have been overly useful to the technicians. However, for a continuous monitoring setup, it would be extremely beneficial, showing the evolution of a frequency component through time.

Appendix A

Communication Protocols

The specifics of the protocols between the sensors and the DAU are described here. Parts of these protocols have been presented throughout the thesis, but all are presented here for completeness.

A.1 Control Protocols

The total set of control characters, first presented in section 5.2.9 are presented again here.

Character	Description
{	Start sending data
}	Stop sending data
<	Get battery level
101010XX	G-select byte mask
?	Read EEPROM
#	Write EEPROM

Table A.1: Valid Control Characters

All actions are initiated by the DAU by sending a single byte to a sensor. The valid bytes are shown in the table.

No matter what byte is sent by the DAU to the sensor, whether or not it's a control byte, the sensor will *always* echo that byte back to the DAU before taking action. This is a simple mechanism to ensure that the byte arrived properly

at the sensor, and gives an easy way to test that the communication channel is functioning correctly.

A.1.1 "Start sending data" Message

The first control character is the "Start sending data" message {.

Upon receiving this byte, the sensor will immediately begin sampling the accelerometer and transmitting data back to the DAU via the packet format described in section 5.2.10. The sensor will continue to transmit until the "Stop sending data" message } byte is received.

While transmitting, all other bytes will be ignored. Not even the echo mechanism previously described is available, the sensor simple ignores all other bytes except for }.

A.1.2 "Stop sending data" Message

When the sensor is sending data, a } byte will immediately stop the transfer and return the sensor to a waiting state. If the sensor is already in a waiting state and this byte is received, the sensor will simply echo it back, taking no additional action.

A.1.3 "Get battery level" Message

The battery level character < tells the sensor to check its current battery level (reading the voltage coming from the batteries at a point before the pump-charger). The value is returned as an 8-bit unsigned integer representing the raw value from the analog-to-digital converter. It is the responsibility of the DAU to interpret this raw value as a percentage of total battery remaining.

It should be noted that only an 8-bit value is returned here, despite the analog-to-digital converter being a 12-bit unit. As very high precision is not needed for the battery level reading the sensor simply sends the eight most-significant-bits of the 12-bit value, throwing away the bottom four bits.

A.1.4 G-Select Byte Mask

The G-select byte mask character is actually a set of four possible characters, used for G selection. These characters tell the sensor to put the accelerometer in

one of its four G modes, 2.5G, 3.3G, 6.7G or 10G. The first six bits of the character, '101010' are used as an identifier mask. If the incoming character matches those six bits exactly (logical AND the incoming character with '10101000' followed by a logical XOR), then the remaining two bits are used to determine which of the four G modes to use.

In particular, the representations are:

Byte	G-mode
10101000	2.5
10101010	3.3
10101001	6.7
10101011	10.0

Table A.2: G-Select Mask

This message will only be interpreted by the sensor as a G-select message if the sensor is in the waiting state.

A.1.5 Read/Write EEPROM

The two EEPROM commands are associated with the reading and writing of calibration data stored in the sensor's EEPROM. At the factory each sensor goes through the calibration procedure described in section 5.3.5, and the individual scaling factors associated with that sensor are stored on the sensor itself. Storing the calibration values on the sensor rather than a DAU means that any sensor can be used with any DAU, the unit just has to query the sensor for its calibration values before data collection begins.

These two EEPROM commands do involve further control characters and very specific send and receive procedures, and these are described fully in Appendix A.2.

A.2 Calibration Protocol

Calibrating the accelerometer on a sensor requires generating calibration values for each of the three axes, for each of the four possible G-ranges that the sensor is capable of operating in. Each axis generates a slope and offset value, each being 32-bit (4-byte) floating point values, so for each G-range a total of

Listing A.1: Calibration Protocol Network Message

```
<''#''><g_value_id><x_slope><x_off><y_slope><y_off><z_slope
  ><z_off>
```

six floating point values need to be stored requiring $6 \cdot 4 = 24$bytes. With four G-ranges available, a total of 96-bytes of storage are required on the sensor.

The values themselves are stored into the EEPROM of the 18F2523. The 18F2523 provides both Flash and EEPROM storage. EEPROM is suitable for long-term storage of program data [45] and is somewhat simpler to access via software than the Flash memory of the 18F2523. The 18F2523 comes equipped with 256-bytes of EEPROM.

A.2.1 Writing Calibration Values

A method for writing the calibration values to the sensor and reading them from the sensor has been devised. It requires the sensor to be fully booted into the normal software to be used.

The generates values for a particular G-range are sent to the sensor at once, so four separate messages must be sent to the sensor to store all four G-ranges. A single message looks as follows:

The first character "#" is simply the unique character used to determine the type of message (as described in section 5.2.9). `g_value_id` is the G-value identifier, telling the sensor which of the four possible G-ranges is being sent. These break down as shown in table A.3.

G-Value	`g_value_id`
2.5G	0
3.3G	1
6.7G	2
10.0G	3

Table A.3: G-Value Identifier Encoding

The values for `g_value_id` must be properly serialized before transmission. For example, in the Python language you would do `chr(2)` for 6.7G.

Listing A.2: Calibration Serialization

```
struct.pack(''!ffffff'', x_slope, x_off, y_slope, y_off,
    z_slope, z_off)
```

The six remaining values in the message all represent four-byte floating point values. These must be transmitted as binary values in network byte order. Using Python again as an example, if each of the six values were stored as Python floats, the serialized data to transmit would be

After the data is stored on the sensor, the sensor will send back another "#" character, to let us know that its done. In total, this means you will receive two "#" characters when you send this. The first one is simply echoing the "#" at the beginning of the message (like all things get echoed by the sensor), the second one is to tell us the EEPROM writing is done.

A.2.2 Reading Calibration Values

Reading back stored calibration values is relatively simple. Transmit two characters, the first being "?" to identify the type of message, and the second being an appropriate `g_value_id`, encoded the same as in table A.3. The sensor will echo back the "?", and then transmit the 24 bytes, in the same order as they would have originally been sent to the sensor.

Appendix B

Sensor Software Configuration

B.1 Timer0 Configuration

The Timer0 module of the 18F2523 is used to generate an interrupt every $2ms$, so the software can perform analog-to-digital conversion at 500Hz.

Configuring the Timer0 module can be split into two main sections, calculating the pre-load values and performing the actual initialization of the necessary registers.

B.1.1 Calculating Timer0 Pre-Load Values

Timer0 operates by incrementing an 8 or 16-bit counter variable every instruction cycle (or every few cycles, as discussed below), and raising an interrupt when the counter overflows. Typically there is a specific amount of time one wishes to wait before the interrupt is generated, and to accommodate this the counter can be pre-loaded before it starts, so it does not have to count through the entire 8 or 16-bit range.

Depending on the desired accuracy or length of time, the user can select to operate Timer0 in 8 or 16-bit modes. When used in 8-bit mode, only the `TMR0L` counter register is used, while 16-bit mode requires both the `TMR0L` (low) and `TMR0H` (high) to be used. These are not the actual registers directly used by the hardware, but instead represent buffered versions of the true high and low bytes of Timer0, which are not directly writable or readable via software.

When choosing the pre-load values, one must take into account more than just the values in these registers. The other factor is the prescaler. This is an

8-bit software controllable register which determines how often the counters will actually be updated. It can be configured for $1:2, 1:4, 1:8, \ldots, 1:256$. In 1:2 mode, the counter will update once per every two full instructions. In 1:4 mode it will update once per every four full instructions, and so on. This prescaler can be used to create very long counter intervals.

For highest accuracy, the sensors were configured with the 16-bit timer and 1:2 prescaler.

Before showing how to configure the timer in software, it is important to show the necessary calculations for choosing the prescale values.

The sensor is equipped with a 20MHz oscillator, i.e. 20 million clock cycles per second. The PIC architecture requires four full clock cycles to perform one complete instruction, so this 20MHz oscillator allows for 5 million instructions per second.

$$\frac{1s}{5 \times 10^6 \text{ instructions}} = 2 \times 10^{-7} \tag{B.1}$$

$$= 200 \times 10^{-9} \tag{B.2}$$

$$= 200 \text{ nanoseconds/instruction} \tag{B.3}$$

With no prescale, overflowing a 16-bit counter (65536 values, requiring 65536 instruction cycles) requires

$$65535 \text{ instructions} \cdot 200 \times 10^{-9} = 13.1072ms \tag{B.4}$$

With the 1:2 prescaler, the counter will increment every

$$2 \cdot 200 \times 10^{-9} = 4 \times 10^{-7} = 0.4\mu s$$

Using the 1:2 prescaler, and a desired $2ms$ interrupt period, the counter would need to be incremented

$$\frac{2 \times 10^{-3}}{4 \times 10^{-7}} = 5000$$

5000 counter increments from a 16-bit counter means the counter must be pre-loaded to

$$65536 - 5000 = 60536$$

The 16-bit counter is split into the TMR0H and TMR0L mentioned above. The division of 60536 into the split counters is easily found with bitwise operations

$$\texttt{TMR0H} = (60536 >> 8) = 236 \tag{B.5}$$

$$\texttt{TMR0L} = 60536 \;\&\; 0xFF = 120 \tag{B.6}$$

B.1.2 Initializing Timer0

Once the appropriate prescale, timer size and pre-load values have been determined, the Timer0 module can be properly initialized in software. This is done in seven distinct steps:

1. Clear TMR0L and TMR0H
2. Set the prescaler and timer size
3. Write pre-load values into TMR0L and TMR0H
4. Turn on Timer0
5. Set Timer0 interrupt priority level
6. Clear Timer0 interrupt flag
7. Enable Timer0 interrupt

Lines 1 and 2 in Listing B.1 simply clear the TMR0L and TMR0H registers. The prescaler and timer size are both set within the T0CON register, and for a 16-bit timer and 1:2 prescale, the appropriate bits within that register are both set to 0. Lines 4 and 5 perform this step.

The preload values (236 and 120) are defined as constants T_HIG and T_LOW, and are loaded and stored into the appropriate registers in lines 7 through 10.

Line 12 sets the TMR0ON bit in the T0CON register to 1, turning on Timer0. Line 13 sets the Timer0 interrupt to be low priority.

Line 15 ensures that the Timer0 interrupt flag (TMR0IF) is cleared in the global interrupt flag tracker, INTCON. If this is not not before enabling interrupts for Timer0, then there is a chance an interrupt will be raised immediately upon enabling the interrupt.

Finally line 16 enables the Timer0 interrupt.

Listing B.1: Timer0 Initialization

```
1  clrf     TMR0L
2  clrf     TMR0H
3
4  movlw    B'00000000'
5  movwf    T0CON
6
7  movlw    T_HIG
8  movwf    TMR0H
9  movlw    T_LOW
10 movwf    TMR0L
11
12 bsf      T0CON,TMR0ON
13 bcf      INTCON2,TMR0IP
14
15 bcf      INTCON,TMR0IF
16 bsf  INTCON,TMR0IE
```

Appendix C

System Identification

While the work presented has its utility in aiding a technician at a site with VA, an eye must be kept on future applicability of the sensors and DAU.

A long term goal is to adapt the system for a permanently installed monitoring situation. This would allow the sensors to run continuously and have the system constantly looking for faults and changes.

A technique that identified when the overall state of the system changed would be beneficial in this situation. It may not (currently) be able to directly interpret what the state change means, but could at least provide a means of automatically notifying a trained technician that *some* change has occurred.

"System Identification" is a technique for automatically identifying the defining characteristics of a system. One tool used for this is *adaptive filters* [7]. They provide a means for generating the representative coefficients of a difference equation of a system without any real previous knowledge of the mechanics of the system, or even a guess as to what the coefficients might be.

The topic of adaptive filters is new to most readers, so an introduction to the structure of adaptive filters will be given, followed by introductions to the characteristics of adaptive filters, the optimal but unrealizable Wiener Filter, and algorithms that attempt to iteratively achieve the Wiener solution. Finally simulations will be presented showing how adaptive filters might be used in conjunction with a vibrating screen.

One of the best introductions to adaptive filters is by Bose [7]. Most of the content presented here is a summary of the adaptive filters introduction from that text.

This section presents an introduction to our work related to using a similar vibration analysis system for long-term continuous monitoring. This work is not

yet complete, but it is our hope that it provides a reasonable starting point for anyone hoping to research continuous vibration analysis systems. In particular, testing the developed system identification software libraries with data from real vibrating screens would always cause severe numerical instability in the results. While we were not able to track down the cause of this ourselves, our hope is that someone with a strong background in system identification might be able to solve the issues and apply this work to continuous monitoring systems.

C.1 Goal

The goal of using system identification techniques should be to provide an automated means of characterizing the mathematical model of a system. With such a model in place, it should be possible to automatically identify faults occurring in the system. The current disadvantage of this technique when applied to vibrating screens is that no existing mathematical model exists for the screens. So there will be no way to take the coefficients for the generated difference equation and translate them directly to particular faults or errors.

What must be done in the future is "break tests", taking vibrating screens known to be fault free, and purposely causing faults in them. The results of system identification for each different break will be instrumental in understanding how the model of a screen changes with particular faults.

Of course this brings back the present problem of multiple screens from multiple manufacturers existing out in the field. There will be no way to perform break tests on every kind of screen a technician might possibly encounter.

Instead, the topic of system identification should probably be limited to future generations of vibrating screens. With knowledge of system identification principles, these screens could be designed with permanently installed sensors, and go through a battery of break tests in the design stages.

Even without break tests and a mathematical model, there is still benefit in using system identification over the long term. The coefficients generated from the identification uniquely identify the current state of the system. A change in time of these coefficients indicates that the state of the system has changed, and this alone is useful information.

Often times a technician is not called to a site until a fault has visibly occurred. Permanently mounted sensors which continuously calculate the coefficients for the system's difference equation would provide a way to see that the state of the system is changing, even before a change is visible from simply looking at the

screen.

C.2 Adaptive Filter Structure

In a digital filter, compensation for a changing system is done by varying the coefficients of the filter, appropriate to the changes occurring in the system. This is done by coupling a digital filter with an adaptive algorithm. The basic filter structure is shown in figure C.1. A desired signal $d(n)$ is fed into the system. The signal $x(n)$ is the input to the filter, and $y(n)$ is the output. The signal $e(n)$ is called the estimation error. This estimation error is fed into the adaptive algorithm, which adjusts the coefficients of the filter, in an attempt to minimize the estimation error.

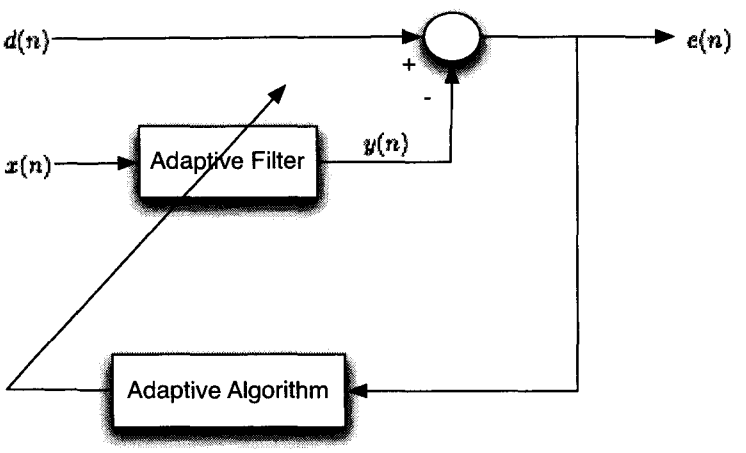

Figure C.1: Adaptive Filter Structure

Many different adaptive filtering schemes exist, with two major points of difference between them:

- Source of the desired signal $d(n)$
- The actual adaptive algorithm

As an example of one possible scheme, a sinusoid tracker is shown in figure C.2. In this system, a sinusoid enters the system with noise covering a wide frequency spectrum. The goal of the system is to extract the sinusoid.

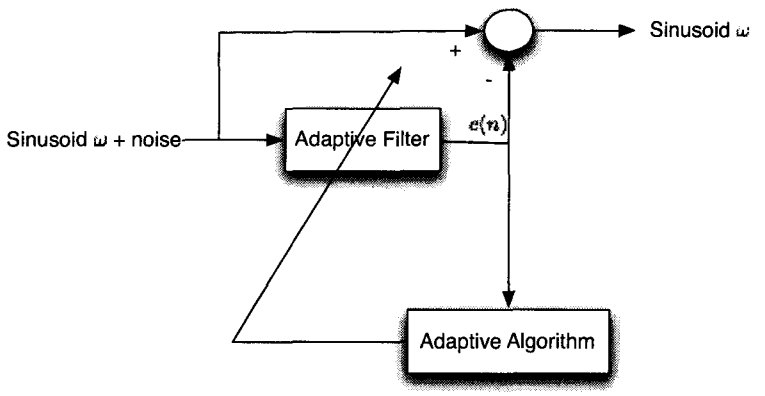

Figure C.2: Sinusoid Tracker

In this system, the adaptive filter used is a notch filter, filtering out all frequencies except for ω, the frequency of the sinusoid. This gives the error signal $e(n)$. If this error signal is subtracted from the original input to the system, then just the sinusoid of frequency ω remains. The objective function of the adaptive algorithm is minimized when the target frequency of the notch filter is ω, resulting in $e(n)$ approximately equal to noise.

If ω were to change over time, the adaptive algorithm would automatically update the notch frequency of the filter, ensuring that the system is still able to output just the sinusoid.

C.3 Characteristics of Adaptive Filters

There are a large variety of adaptive filter algorithms described in the literature, but in general there are four characteristics to be considered with when evaluating an adaptive algorithm:

1. Computational complexity
2. Rate of convergence
3. Numerical robustness
4. Misadjustment

The first three of these are fairly typical when analyzing any form of numerical algorithm. The importance of computational complexity and the rate of convergence of the algorithm greatly depend on the target system. If the system can change quickly, and requires a high sampling rate, then computational complexity and rate of convergence will both be very important. There is a direct relationship between computational complexity and sampling rate. The higher the required sampling rate, the lower we require the computational complexity to be. Some systems might require a high sampling rate, but changes occur very slowly. In that case, we would be concerned with the computational complexity, but not necessarily with the rate of convergence.

As mentioned in section C.2, the goal of the adaptive algorithm is to minimize the objective function, related to the error $e(n)$. Quite often, the objective function to be minimized is the mean square error (MSE), given by $E\{e^2(n)\}$.

The Wiener algorithm is the theoretical optimal solution for this objective function. As will be discussed in section C.4, the Wiener algorithm is not practical for real world implementations. Instead, the goal is usually to develop an algorithm that attempt to approach the Wiener solution. If J_{opt} is the MSE obtained by the Wiener algorithm, and J_{ss} is the MSE obtained by some other adaptive algorithm, then $J_{ex} = J_{opt} - J_{ss}$ is the Excess MSE. The misadjustment is simply J_{ex}/J_{opt}. Clearly, a lower misadjustment is better.

C.4 Wiener Algorithm

The Wiener Filter [73] is the optimal, but unrealizable filter that adaptive filter theory is based on. The basic problem is illustrated in figure C.3.

This is the same basic structure as we've seen before. The input $x(n)$ is passed through a filter with coefficients $w_0, w_1, \ldots, w_{N-1}$, giving an output $y(n)$. The goal is to find the coefficients that minimize the MSE. The output $y(n)$ is that of a basic FIR filter, namely:

$$\begin{aligned} y(n) &= w_0 x(n) + w_1 x(n-1) + \ldots + w_{N-1} x(n-N+1) \\ &= \vec{w}^T \vec{x}(n) \end{aligned}$$

Our goal is to minimize the objection function, which will be

$$J(\vec{w}) = E\{e^2(n)\}$$

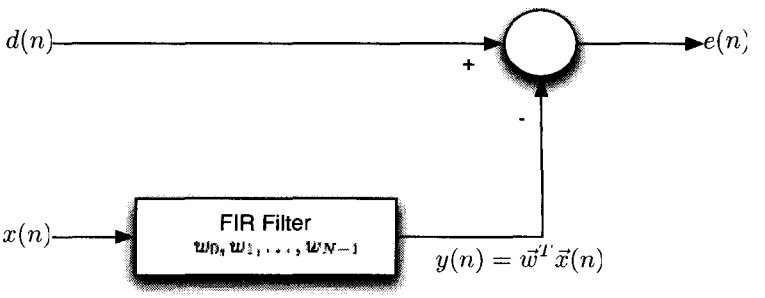

Figure C.3: Wiener Filtering

Expanding this gives us:

$$\begin{aligned} J(\vec{w}) &= E\{e^2(n)\} && \text{(C.1)} \\ &= E\{[d(n)-y(n)]^2\} && \text{(C.2)} \\ &= E\{d^2(n)-2d(n)y(n)+y^2(n)\} && \text{(C.3)} \end{aligned}$$

Substituting in $y(n) = \vec{w}^T\vec{x}(n)$, gives:

$$\begin{aligned} J(\vec{w}) &= E\{d^2(n)-2d(n)y(n)+(\vec{w}^T\vec{x}(n))^2\} \\ &= E\{d^2(n)-2d(n)\vec{w}^T\vec{x}(n)+\vec{w}^T\vec{x}(n)\vec{x}^T(n)\vec{w}\} \end{aligned}$$

The filter coefficients $\vec{w}$ are not random variables, so they can be pulled out of the expectation operator:

$$J(\vec{w}) = E\{d^2(n)\}-2\vec{w}^T E\{d(n)\vec{x}(n)\}+\vec{w}^T E\{\vec{x}(n)\vec{x}^T(n)\}\vec{w}$$

We can apply some simplifications to the cost function, to aid in understanding it.

If we assume that $d(n)$ is zero mean (i.e. $\mu = 0$) then $E\{d^2(n)\}$ is actually σ_d^2, the variance of $d(n)$.

$$E\{d^2(n)\} = E\{|d(n)-\mu|^2\} \tag{C.4}$$
$$= E\{|d(n)-0|^2\} \tag{C.5}$$
$$= \sigma_d^2 \tag{C.6}$$

Define $\vec{p} \equiv E\{d(n)\vec{x}(n)\} = [p(0), p(1), \ldots, p(N-1)]^T$ as the cross-correlation between the desired signal $d(n)$ and the input signal $\vec{x}(n)$. In the third term of the cost function, we have $E\{\vec{x}(n)\vec{x}^T(n)\}$, which is actually the correlation matrix $\mathbf{R}$ [74].

We can now rewrite the cost function as:

$$J(\vec{w}) = \sigma_d^2 - 2\vec{w}^T\vec{p} + \vec{w}^T\mathbf{R}\vec{w}$$

The second term is linear, and the third is quadratic, or convex. Since convex functions have minimum points (found by taking the gradient w.r.t. $\vec{w}$ and setting to zero), we can easily solve it.

As an example, let us examine a 2-tap Wiener filter. A 2-tap filter will have two elements in the w vector, namely $\vec{w} = [w_0, w_1]$. So we want to solve $J(w_0, w_1)$. The objective function is defined as follows:

$$\begin{aligned} J(w_0,w_1) &= E\{d^2(n)\} - 2E\{d(n)y(n)\} + E\{y^2(n)\} \\ &= \sigma_d^2 - 2E\{d(n)[w_0x(n) + w_1x(n-1)]\} \\ &\quad + E\{[w_0x(n) + w_1x(n-1)]^2\} \\ &= \sigma_d^2 - 2w_0E\{d(n)x(n)\} - 2w_1E\{d(n)x(n-1)\} \\ &\quad + w_0^2E\{x^2(n)\} + 2w_0w_1E\{x(n)x(n-1)\} \\ &\quad + w_1^2E\{x^2(n-1\} \end{aligned} \tag{C.7}$$

This simplifies to the Wiener objective function

$$J(w_0,w_1) = \sigma_d^2 - 2w_0p(0) - 2_1p(1) + w_0^2r(0) + 2w_0w_1r(1) + w_1^2r(0) \tag{C.8}$$

Now take the partial derivative, w.r.t. w_0 and w_1:

$$\frac{\partial J}{\partial w_0} = -2p(0) + 2w_0r(0) + w_1r(1) \tag{C.9}$$

$$\frac{\partial J}{\partial w_1} = -2p(1) + 2w_0 r(1) + w_1 r(0) \tag{C.10}$$

The gradient of the cost function can now be written as

$$\nabla J(w_0, w_1) = \begin{bmatrix} \frac{\partial J}{\partial w_0} \\ \frac{\partial J}{\partial w_1} \end{bmatrix} \tag{C.11}$$

$$= -2\begin{bmatrix} p(0) \\ p(1) \end{bmatrix} + 2\begin{bmatrix} r(0) & r(1) \\ r(1) & r(0) \end{bmatrix}\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \tag{C.12}$$

So for the general case $(N-1)$ tap filter, we get:

$$\nabla J(\vec{w}) = -2\vec{p} + 2\mathbf{R}\vec{w} \tag{C.13}$$

Set this to 0 and solve for $\vec{w}$ to get the Wiener filter:

$$\vec{w} = \mathbf{R}^{-1}\vec{p} \tag{C.14}$$

$\vec{w}$ is the optimal coefficient vector for the filter. In reality though, this is infeasible, as computing $\mathbf{R}^{-1}$ is too computationally expensive.

Fortunately there are algorithms that do quite a good job of approaching the Wiener solution (low misadjustment), via an iterative method, while minimizing the computational overhead.

The first class of such algorithms are based on a gradient search, or Newton's algorithm. It can be shown [7] that Newton's algorithm can converge to the Wiener solution in one step. However, the computation requires performing an inverse on a Hessian, and is thus still infeasible. For the sake of reference, the Newton algorithm is

$$\vec{w}(n+1) = \vec{w}(n) - \mathbf{H}^{-1}\nabla\mathbf{J} \tag{C.15}$$

The Steepest Descent algorithm, shown in Equation C.16 uses an approximation of $\mathbf{H} = 2\mathbf{I}$ for the Hessian, allowing for a computationally feasible approximation of Wiener's algorithm. The value μ is introduced to control the rate of convergence.

$$\vec{w}(n+1) = \vec{w}(n) - \frac{\mu}{2}\nabla\mathbf{J} \tag{C.16}$$

For adaptive filtering, we can substitute for the gradient $\nabla \mathbf{J}$ with the value from Equation C.13 to give:

$$\vec{w}(n+1) = \vec{w}(n) + \mu[\vec{p} - \mathbf{R}\vec{w}(n)] \tag{C.17}$$

Unfortunately, for many real-time applications the Steepest Descent algorithm is still infeasible. Note the presence of $\vec{p}$ and $\mathbf{R}$. We defined both of these values via the expectation operator $E\{\cdot\}$. In many real-time systems, only a single realization of the signal is available, thus $\vec{p}$ and $\mathbf{R}$ must be estimated.

C.4.1 Least Mean Squares

The Least Mean Squares (LMS) algorithm is a popular means for implementing Steepest Descent with estimations for $\vec{p}$ and $\mathbf{R}$. The estimations are given as

$$R = E\{\vec{x}(n)\vec{x}^T(n)\} \tag{C.18}$$
$$\simeq \vec{x}(n)\vec{x}^T(n) \tag{C.19}$$

and

$$p = E\{d(n)\vec{x}^T(n)\} \tag{C.20}$$
$$\simeq d(n)\vec{x}^T(n) \tag{C.21}$$

Substituting these into Equation C.17 and simplifying, we get:

$$\vec{w}(n+1) = \vec{w}(n) - \mu\vec{x}(n)e(n) \tag{C.22}$$

This is the LMS algorithm, first introduced by Widrow and Hoff [75].

The LMS is quite popular for both its simplicity in implementation and a low $O(N)$ computational complexity, where N is the number of taps in the filter.

While LMS is popular for real-time systems, it does have a few drawbacks. Namely, it can never have a misadjustment of zero (i.e. it cannot reach the Wiener filter). The larger the step size, the higher the misadjustment, but the larger the step size, the faster the rate of convergence. So a step size must be chosen that allows for the best trade off.

C.4.2 Recursive Least Squares

There is a separate class of algorithms called least-squares algorithms which are not as popular for high speed applications (thanks to a relatively high computational complexity), but are much better than LMS when it comes to misadjustment and convergence speed. The most popular is the Recursive Least Squares (RLS). RLS approaches the optimal weight vector solution in a finite number of steps, while LMS approaches as the number of iterations approaches infinity. Also, the RLS MSE tends towards zero, while LMS can never achieve this. The RLS though has a complexity of $O(N^2)$ [7].

C.4.3 Experimental Comparison of LMS and RLS

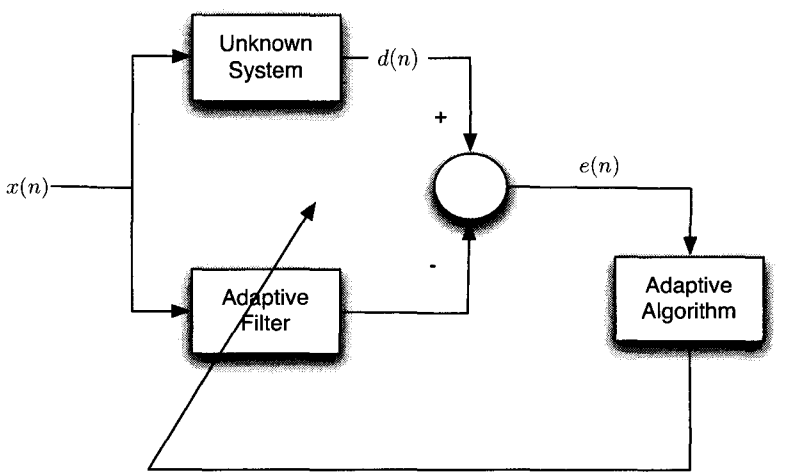

Figure C.4: System Identification

As a comparison, we have built implementations of both the LMS and RLS algorithms. To test them, a system identification experiment proposed in [7] was used. This problem is diagrammed in figure C.4.

In short, we started with an FIR filter given by

$$g(n) = 0.1x(n) + 0.2x(n-1) + 0.3x(n-2)$$

We want to add some noise to the output of this, and attempt to have an adaptive filter discover and match the coefficients.

The input signal $x(n)$ is a white Gaussian sequence with unit variance. The output is generated using the FIR equation $g(n)$, and then added to white Gaussian noise, giving the "desired signal" $d(n)$. The same input signal is fed into the Adaptive Filter. The adaptive algorithm attempts to minimize the difference between the Unknown System and the Adaptive Filter. The best it can do is match the coefficients of the Adaptive Filter and the original FIR of the system, with the eventual error signal being just the white Gaussian noise that was added to the output of the FIR.

The results of this experiment for an LMS filter and an RLS filter are shown in figures C.5 and C.6, respectively. Notice how little variance around the coefficients there is in the RLS case, vs LMS. Also notice that the RLS experiment converged in far fewer steps than the LMS algorithm. However, for a 300 sample test, the RLS algorithm was thirty times slower.

Each experiment was run one hundred times to get an ensemble of data.

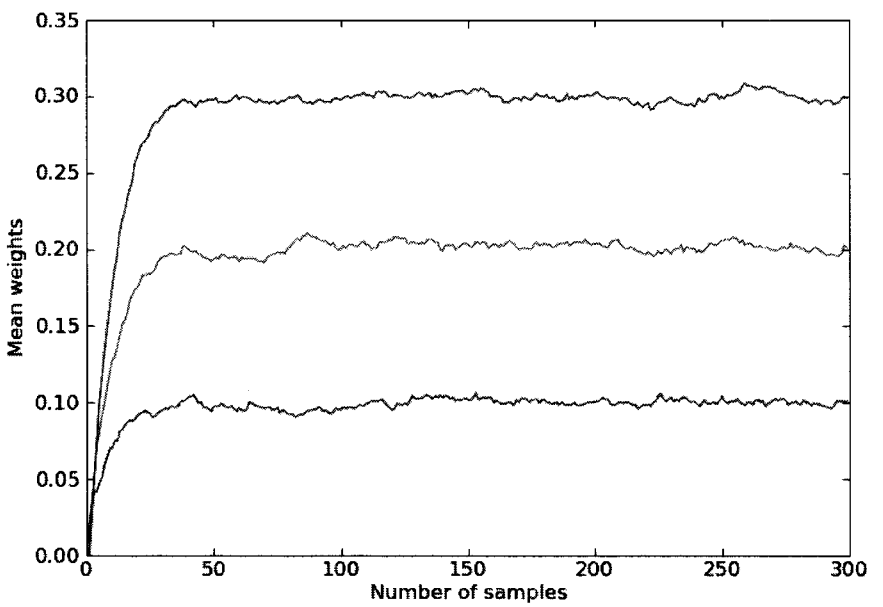

Figure C.5: LMS System Identification

A faster version of RLS has recently been developed, called Euclidean Direction Search (EDS), which combines the best of both RLS and LMS [76] . EDS

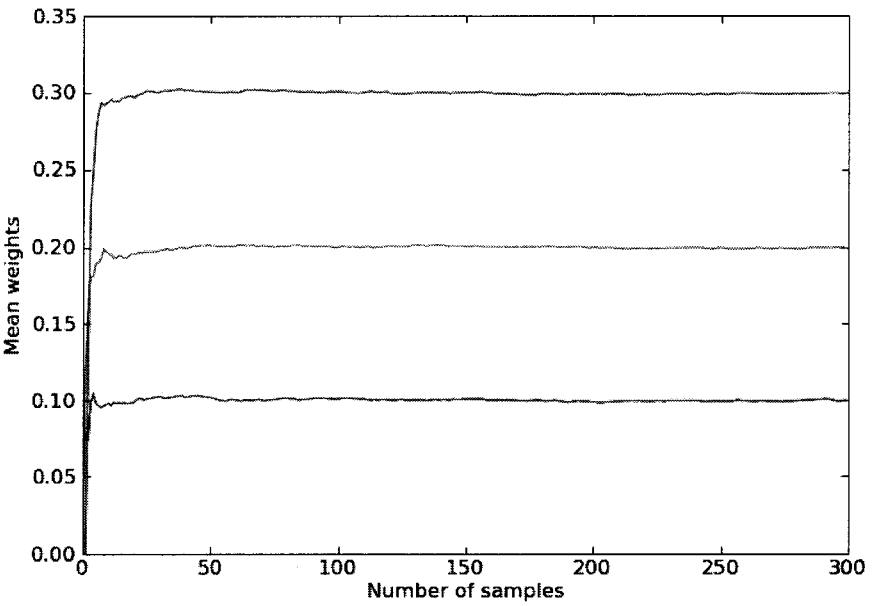

Figure C.6: RLS System Identification

is still $O(N^2)$, but the computational complexity lands somewhere between LMS and RLS. It has been proved that the EDS algorithm converges to the optimal solution [77].

C.5 Adaptive Filters for System Identification of Vibrating Screens

Returning to the original problem, how can these adaptive filters be used to identify the state of a screen, and more importantly, how can they be used to identify *changes* to the state over time?

Referring again to figure C.4, the first question is where $x(n)$ input signal would come from. Ideally this would be a signal directly from the motor driving the screen, so the driving force of the motor could be used to directly generate the state of the system.

Unfortunately on the current generation of screens, this is just not possible.

There is too much variability in terms of the types of motors, and most have no way to connect an output channel for monitoring the signal generated from the motor.

Instead, the input signal from the motor must be determined indirectly.

On a correctly functioning screen, the driving force of the motor is directly proportional to the main operating frequency and motion of the running screen. Using the Butterworth-filtered output of the signal from the sensors, all harmonic content is eliminated except for the main driving force of the machine. Thus the output of the Butterworth filter can be interpreted as the input $x(n)$ of figure C.4.

Essentially we propose to take the filtered values as $x(n)$ and use the unfiltered values as $d(n)$, the desired value the adaptive filter should attempt to reach. So the unfiltered values are the result of an unknown system being driven by an input defined by the filtered values.

The main assumption being made here is that interesting changes to the state of the system will take place outside of the operating frequency of the screen. To guarantee this assumption holds, a check will be required before running through the adaptive filter.

Before passing the $x(n)$ values through, it must be verified that the frequency and amplitude of $x(n)$ has not changed since the last time this experiment was performed. Even in a continuous monitoring situation, the system state will most likely not be checked continuously. Instead it would probably be checked periodically, say every few hours. Every time a new system identification period begins, the system will first check that neither the frequency nor amplitude of $x(n)$ have changed. If they remain the same, it means the assumed input signal remains the same.

As an aside, if either the frequency of amplitude *have* changed, then something more fundamental is going wrong. Frequency and amplitude are just the mathematical bases for RPM and stroke, two characteristics that technicians already look at. If either of these values are outside of a normal range, then those problems need to be addressed first.

Assuming that $x(n)$ has not changed, then the filtered and unfiltered values can be used with the adaptive filter, to generate system coefficients within the filter itself. If these coefficients have changed since the last time the test was run, then some factor of the system state has changed and the technician should investigate further.

While the end-goal would be to implement these systems on permanently installed screens, there is a benefit to performing system identification with the system designed for this thesis. Technicians are often sent to a site to look at one

problematic screen, sites where there are sometimes dozens of screens present. Because a VA session can be completed very quickly with the new system, it might be worth the effort to have the technician perform VA on every screen, each time they are present at the site.

This would be useful for preventative fault detection. If the state of the system has changed, then some component of the screen's operation has changed, and even if nothing appears to have gone wrong yet, it would provide enough evidence to warrant further investigation by the technician while they are at the site. Faults detected early are invariably cheaper to repair than faults detected only after a problem has already occurred, and it reduces the number of trips a technician might have to make to an individual site.

Bibliography

[1] Stephan Ebersbach and Zhongxiao Peng. Expert system development for vibration analysis in machine condition monitoring. *Expert Syst. Appl.*, 34(1):291–299, 2008.

[2] Andrew K.S. Jardine, Daming Lin, and Dragan Banjevic. A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical Systems and Signal Processing*, 20(7):1483 – 1510, 2006.

[3] Kenneth A. Loparo Hasan Ocak and Fred M. Discenzo. Online tracking of bearing wear using wavelet packet decomposition and probabilistic modeling: A method for bearing prognostics. *Journal of Sound and Vibration*, 302(4-5):951–961, 2007.

[4] Trygve Reenskaug. The model-view-controller (mvc) its past and present. 2003.

[5] . Smith, J. D. *Vibration measurement and analysis / J.D. Smith.* Butterworths, London :, 1989.

[6] Julius O. Smith. *Introduction to Digital Filters with Audio Applications.* W3K Publishing, `http://www.w3k.org/books/`, 2007.

[7] Tamal Bose and Francois Meyer. *Digital Signal and Image Processing.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[8] Steven W. Smith. *The scientist and engineer's guide to digital signal processing.* California Technical Publishing, San Diego, CA, USA, 1997.

[9] Travis E. Oliphant. *Guide to NumPy*, 2006.

[10] Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, second edition, 2004.

[11] Maple - math and engineering software. `http://www.maplesoft.com/products/Maple/index.aspx`.

[12] Bin Zhang, G. Georgoulas, M. Orchard, A. Saxena, D. Brown, G. Vachtsevanos, and S. Liang. Rolling element bearing feature extraction and anomaly detection based on vibration monitoring. pages 1792 –1797, jun. 2008.

[13] Ian. Howard, I. M. Howard, Defence Science, Technology Organisation (Australia), Aeronautical, and Maritime Research Laboratory (Australia). *A review of rolling element bearing vibration : detection, diagnosis and prognosis / Ian Howard*. DSTO Aeronautical and Maritime Research Laboratory, Melbourne :, 1994.

[14] D. F. Shi, W. J. Wang, and L. S. Qu. Defect detection for bearings using envelope spectra of wavelet transform. *Journal of Vibration and Acoustics*, 126(4):567–573, 2004.

[15] Shahab Hasanzadeh Ghafari. *A Fault Diagnosis System for Rotary Machinery Supported by Rolling Element Bearings*. PhD thesis, University of Waterloo, 2007.

[16] Changzheng Chen and Changtao Mo. A method for intelligent fault diagnosis of rotating machinery. *Digital Signal Processing*, 14(3):203–217, 2004.

[17] Bo-Suk Yang, Dong-Soo Lim, and Andy Chit Chiow Tan. Vibex: an expert system for vibration fault diagnosis of rotating machinery using decision tree and decision table. *Expert Syst. Appl.*, 28(4):735–742, 2005.

[18] CHUEI-TIN CHANG, KAI-NAN MAH, and CHII-SHIANG TSAI. A simple design strategy for fault monitoring systems, 1993.

[19] S. Edwards, A. W. Lees, and M.I. Friswell. Fault diagnosis of rotating machinery. *Shock and Vibration Digest*, 1998.

[20] Jesús Manuel, Jesús Manuel Fernández Salido, Advising Professor, and Shuta Murakami. Design of a diagnosis system for rotating machinery using fuzzy pattern matching and genetic algorithms, 1998.

[21] Yaguo Lei, Zhengjia He, and Yanyang Zi. A new approach to intelligent fault diagnosis of rotating machinery. *Expert Systems with Applications*, 35(4):1593 – 1600, 2008.

[22] Qiao Sun, Ping Chen, Dajun Zhang, and Fengfeng Xi. Pattern recognition for automatic machinery fault diagnosis. *Journal of Vibration and Acoustics*, 126(2):307–316, 2004.

[23] Scott W. Doebling, Charles R. Farrar, and Michael B. Prime. A summary review of vibration-based damage identification methods. *Identification Methods," The Shock and Vibration Digest*, 30:91–105, 1998.

[24] JING LIN and LIANGSHENG QU. Feature extraction based on morlet wavelet and its application for mechanical fault diagnosis. *Journal of Sound and Vibration*, 234(1):135 – 148, 2000.

[25] Y. Y. KIM, J. C. HONG, and N. Y. LEE. Frequency response function estimation via a robust wavelet de-noising method. *Journal of Sound and Vibration*, 244(4):635 – 649, 2001.

[26] Jing Lin, Ming J. Zuo, and Ken R. Fyfe. Mechanical fault detection based on the wavelet de-noising technique. *Journal of Vibration and Acoustics*, 126(1):9–16, 2004.

[27] G. K. Singh and Saleh Al Kazzaz Sa'ad Ahmed. Vibration signal analysis using wavelet transform for isolation and identification of electrical faults in induction machine. *Electric Power Systems Research*, 68(2):119 – 136, 2003.

[28] Jou-Wei Lin, A.F. Laine, and S.R. Bergmann. Improving pet-based physiological quantification through methods of wavelet denoising. *Biomedical Engineering, IEEE Transactions on*, 48(2):202 –212, feb. 2001.

[29] L.J. Hadjileontiadis, C.N. Liatsos, C.C. Mavrogiannis, T.A. Rokkas, and S.M. Panas. Enhancement of bowel sounds by wavelet-based filtering. *Biomedical Engineering, IEEE Transactions on*, 47(7):876 –886, jul. 2000.

[30] L.J. Hadjileontiadis. Wavelet-based enhancement of lung and bowel sounds using fractal dimension thresholding-part ii: application results. *Biomedical Engineering, IEEE Transactions on*, 52(6):1050 –1064, jun. 2005.

[31] G. Y. Luo, D. Osypiw, and M. Irle. On-Line Vibration Analysis with Fast Continuous Wavelet Algorithm for Condition Monitoring of Bearing. *Journal of Vibration and Control*, 9(8):931–947, 2003.

[32] Wim Sweldens. The lifting scheme: a construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1998.

[33] Stephane Mallat and Wen Liang Hwang. Singularity detection and processing with wavelets. *IEEE Transactions on Information Theory*, 38:617–643, 1992.

[34] David L. Donoho. De-noising by soft-thresholding, 1992.

[35] Saad Ahmed Saleh Al Kazzaz and G. K. Singh. Experimental investigations on induction machine condition monitoring and fault diagnosis using digital signal processing techniques. *Electric Power Systems Research*, 65(3):197 – 221, 2003.

[36] Erik H. Clayton, Bong hwan Koh, Guoliang Xing, Chien liang Fok, Shirley J. Dyke, and Chenyang Lu. Damage detection and correlation-based localization using wireless mote sensors.

[37] J. Vollmer, Yanting Hu, T. Neumann, and E. Solda. Construction kit for low-cost vibration analysis systems based on low-cost acceleration sensors. pages 463 –468, jul. 2009.

[38] Markos Markou and Sameer Singh. Novelty detection: a review–part 1: statistical approaches. *Signal Processing*, 83(12):2481 – 2497, 2003.

[39] Dipankar Dasgupta and Stephanie Forrest. Novelty detection in time series data using ideas from immunology. In *In Proceedings of The International Conference on Intelligent Systems*, 1995.

[40] Michael Haag. *Crosscorrelation of Random Processes.* Connexions, `http://cnx.org/content/m10686/2.2/`, 2005.

[41] Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT).* W3K Publishing, `http://www.w3k.org/books/`, 2007.

[42] J. Adams and B. Heile. Busy as a zigbee. *Spectrum, IEEE*, 43(10), Oct. 2006.

[43] Bluetooth Special Interest Group. Bluetooth specification version 1.1 and 1.2. http://www.bluetooth.com.

[44] Igor Gurovski and Velimir Karadzic. Self-configuring bluetooth networks. http://www.connectblue.se/fileadmin/Connectblue/PDF/White_papers/Self-configuring_Bluetooth_Networks.pdf.

[45] Microchip. Pic18f2423/2523/4423/4523 data sheet. http://ww1.microchip.com/downloads/en/DeviceDoc/39755c.pdf.

[46] Alan C. Shaw. *Real-Time Systems and Software*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[47] Jinhui Lan, Saeid Nahavandi, Yixin Yin, and Tian Lan. Development of low cost motion-sensing system. *Measurement*, 40(4):415 – 421, 2007.

[48] Microchip. Flash microcontroller programming specification. http://ww1.microchip.com/downloads/en/DeviceDoc/30277d.pdf.

[49] Stefan Behnel, Robert Bradshaw, and Dag Sverre Seljebotn. Cython: C-extensions for python. http://www.cython.org.

[50] David Lorge Parnas. Document based rational software development. *Know.-Based Syst.*, 22(3):132–141, 2009.

[51] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[52] Joe Armstrong. Erlang — software for a concurrent world. In *ECOOP '07: Proceedings of the 21st European conference on ECOOP 2007*, pages 1–1, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] Hsinfu Huang and Hsin-His Lai. Factors influencing the usability of icons in the lcd touchscreen. *Displays*, 29(4):339 – 344, 2008.

[54] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[55] Vivek Sarkar. Optimized unrolling of nested loops. *Int. J. Parallel Program.*, 29(5):545–581, 2001.

[56] The-Scipy-community. scipy.signal.freqz. http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.freqz.html, February 2010.

[57] Tae Hong Park. Salient feature extraction of musical instrument signals. Master's thesis, Dartmouth College, 2000.

[58] Tripod Data Systems. Tds nomad product brochure. http://www.tdsway.com.

[59] ARM Holdings. Arm - the architecture for the digital world. http://www.arm.com.

[60] Linksys by cisco bluetooth usb adapter usbbt100. http://www.linksysbycisco.com/US/en/products/USBBT100.

[61] The ångström distribution. http://www.angstrom-distribution.org/.

[62] The GTK+ Project. The gtk+ project. http://www.gtk.org.

[63] wxwidgets cross-platform gui library. http://www.wxwidgets.org.

[64] Python Software Foundation. Python programming language – official website. http://python.org.

[65] Jeremy Eric Elson, Mario Gerla, Gerald J. Popek, Gregory J. Pottie, Majid Sarrafzadeh, Deborah L. Estrin, and Committee Chair. Time synchronization in wireless sensor networks, 2003.

[66] J. Mannermaa, K. Kalliomaki, T. Mansten, and S. Turunen. Timing performance of various gps receivers. volume 1, pages 287 –290 vol.1, 1999.

[67] Jagoba Arias, Eduardo Santos, Itziar Marin, Jaime Jimenez, Jesus Lazaro, and Aitzol Zuloaga. Node synchronization in wireless sensor networks. *Wireless and Mobile Communications, International Conference on*, 0:50, 2006.

[68] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Continuous clock synchronization in wireless real-time applications. pages 125 –132, 2000.

[69] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, 2002.

[70] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281 – 323, 2005.

[71] R. Casas, H.J. Gracia, A. Marco, and J.L. Falco. Synchronization in wireless sensor networks using bluetooth. pages 79 – 88, may 2005.

[72] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149, New York, NY, USA, 2003. ACM.

[73] Norbert Wiener. *Extrapolation, Interpolation, and Smoothing of Stationary Time Series*. The MIT Press, 1964.

[74] William A. Gardner. *Introduction to Random Processes*. McGraw-Hill, second edition, 1990.

[75] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. pages 123–134, 1988.

[76] G.F. Xu, T. Bose, W. Kober, and J. Thomas. A fast adaptive algorithm for image restoration. *IEEE Transactions on Circuits and Systems*, January 1999.

[77] G.F. Xu. *Fast Algorithms for Digital Filtering: Theory and Applications*. PhD thesis, University of Colorado, Boulder, 1999.