CONDITION MONITORING AND FAULT DETECTION OF BLADE DAMAGE IN SMALL WIND TURBINES USING TIME-SERIES AND FREQUENCY ANALYSES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Luke Hayden Costello

March 2021

COMMITTEE MEMBERSHIP

TITLE:	Condition Monitoring and Fault Detection of Rotating Unbalance in Small Wind Turbines Using Timeseries and Frequency Analysis Methods
AUTHOR:	Luke Hayden Costello
DATE SUBMITTED:	March 2021
COMMITTEE CHAIR:	Dr. Patrick Lemieux, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Dr. John Ridgely, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Dr. Xi Wu, Ph.D. Professor of Mechanical Engineering

ABSTRACT

Condition Monitoring and Fault Detection of Rotating Unbalance in Small Wind Turbines Using Timeseries and Frequency Analysis Methods

Luke Hayden Costello

Condition monitoring systems are critical for autonomous detection of damage when operating remote wind turbines. These systems continually monitor the turbine's operating parameters and detect damage before the turbine fails. Although common in utility-scale turbines, these systems are mostly undeveloped in distributed, small-scale turbines due to their high cost and need for specialized equipment. The Cal Poly Wind Power Research Center is developing a low-cost, modular solution known as the LifeLine system. The previous version contained monitoring equipment, but lacked decision-making capabilities.

The present work builds on the LifeLine by developing software-based detection of blade damage. Detection is done by monitoring of tower vibrations, rotor speed, and generator power output. First, testing is completed to inform algorithm design: the tower vibrational response is recorded, and blade damage is simulated by adding a mass imbalance to one blade. From these results, several algorithms are developed, and their performance is analyzed in a cross-validation study. The time-series method known as the Nonlinear State Estimation Technique and Sequential Probability Ratio Test (NSET+SPRT) is implemented first. This algorithm is highly successful, with a 93.3% rate of correct damage detection; however, it occasionally raises false alarms during normal operation. A custom-built algorithm known as the Adaptive Fast Fourier Transform (AFFT) is also built; its strength lies in its elimination of false alarms. The final system utilizes a joint monitoring approach, combining the benefits of the NSET+SPRT and AFFT. The final algorithm is successful, correctly categorizing 95.5% of data when operating above 120RPM, and raising no false alarms in normal operation. This version is then implemented for live monitoring on the Cal Poly Wind Turbine, allowing for robust and autonomous detection of blade damage.

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Lemieux, Dr. Ridgely, and all other students and faculty that have developed the Cal Poly Wind Turbine. This project would not exist without their work, and for that, I am incredibly grateful. Second, thank you to my family, who supported – mentally and financially – my entire education, up to and including my graduate degree. Finally, thank you to Ryan Zhan and John Cunningham, who accompanied me to the turbine countless times; without them, the heavy amount of field testing necessary for completing this thesis would not have been possible.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

NOMENCLATURE

ACRONYMS

CMS	Condition Monitoring System
COE	Cost of Energy
CPWT	Cal Poly Wind Turbine
FFT	Fast Fourier Transform
NSET	Nonlinear State Estimation Technique
O&M	Operating and Maintenance
RPi	Raspberry Pi Computer
SCADA	Supervisory Control and Data Acquisition System
SPRT	Sequential Probability Ratio Test

VIBRATIONS

RMS	Root Mean Squared
LL	Line Length
CF	Crest Factor
SF	Shape Factor
1P	Vibrations occurring at 1 times the rotor operating speed
3P	Vibrations occurring at 3 times the rotor operating speed

AERODYNAMICS

λ	Tip Speed Ratio [#]
A	Rotor swept area [m^2]
u_∞	Wind Speed [m/s]
V_R	Wind speed relative to airfoil chord [m/s]
ϕ	Angle of relative wind vector
ρ_{air}	Air density [kg/m^3]
Ω	Rotor Speed [RPM]
ω	Rotor Speed [rad/s]
Θ	Angle of a blade from horizontal [rad]
R	Rotor Radius [m]
r	Mass imbalance radius [m]
dr	Differential length of blade section [m]
c	Airfoil Chord Length [m]
C_l	Airfoil Lift Coefficient [#]
C_d	Airfoil Drag Coefficient [#]
P_{mech}	Mechanical power output from wind turbine [kW]

Chapter 1

INTRODUCTION

1.1 Project Motivation

Compared to conventional power generation, wind turbines may operate in extremely remote areas. Wind resources are typically at their highest far from city regions, especially offshore turbines. High wind resources also cause environmental stresses, wearing turbines over time. The combination of remote conditions, continual environmental degradation, and servicing difficulty causes operating and maintenance (O&M) costs to be very high compared to conventional power generation. A 2006 report by Sandia National Laboratory found that O&M costs can account for 10 – 20% of a wind turbine's Cost of Energy (COE) [1]. As Kusiak notes in ref. [2], the replacement of a $5000 bearing may quickly turn into a $250,000 project, due to the work crews and heavy machinery necessary to service the machine. Furthermore, the damage is often only noticed once it significantly impacts operation, at which point once-localized damage may have expanded and impacted other components. Reducing O&M costs and detecting damage early is therefore a major topic of study in improving the cost of wind power. This may be done, in part, by implementing a condition-based monitoring and fault detection system (CMS). Monitoring is usually done via Supervisory Control and Data Acquisition (SCADA) systems, which continuously monitor and record various system parameters.

Fault detection systems are often-used in utility-scale wind turbines; however, they are much less common in small-scale, distributed wind systems. Utility-scale wind typically involves large turbines capable of producing > 1MW per turbine, with power being actively fed into the electric grid [3]. As a result, these systems can and must implement highly reliable (and thus expensive) condition monitoring solutions. On the other hand, small or medium-scale distributed wind typically involves turbines producing < 500 kW [4], with power only being used for local

communities, without a grid connection [5]. As a result, distributed wind systems often cannot justify the implementation of expensive monitoring systems. As these small-scale turbines become more common, there will be an expanding need for a low-cost monitoring solution. To solve these issues, the Cal Poly Wind Power Research Center is developing the LifeLine monitoring system. The goal of the LifeLine project is to create a low-cost, modular condition monitoring system that may be adapted to a wide variety of wind turbines. As such, it requires hardware and software that may be applied to any turbine, rather than any one specifically. This project is currently in its infancy; currently, it consists of a MicroPython-based microcontroller and accelerometer, and simply collects acceleration data. The purpose of the present work is to continue development of the LifeLine system by designing a condition monitoring algorithm using the currently installed sensors. Experience has shown that simple methods for fault detection, such as fixed vibration thresholds, result in a high rate of false positives. This results in unnecessary travel to the turbine site and causes the turbine to be shut down when it could be generating power. Thus, real-time monitoring must strike a balance between sensitivity to damage, and resistance to false alarms.

In addition to the accelerometer, available sensors in the Cal Poly Wind Turbine (CPWT) nacelle include a current and voltage sensor from the generator, a rotor speed sensor, a wind speed sensor, and a wind vane sensor. In consideration of these available sensors, the present work will focus on detecting damage to the turbine blades. In addition to this work, LifeLine development is ongoing via other student projects: see Ryan Zhan's [6] and Ryan Takatsuka's [7] theses for more information.

1.2 Causes and Effects of Blade Damage

In general, blade damage has two potential sources: manufacturing defects, and environmental damages. The most common manufacturing defects, according to ref. [8], are waviness (resulting from improper composite construction) and porosity or voids in the blade structure. Although

significant, these damages may be identified before turbine assembly. Environmental damages, on the other hand, pose a much greater risk to a turbine over its life.

First, blade surface degradation, or roughing, gradually occurs over time for all blades. It is caused by rain, hail, and other debris – but is especially pronounced in corrosive environments, such as offshore and desert locations. As small particulates contact the blade surface, pitting – or small gouges in the blade surface – occur. This is primarily concentrated near the leading edge. Blade surface degradation may be monitored by measuring the power performance of the turbine over its life. Power performance is typically measured using the power coefficient of the turbine, C_p – the ratio of mechanical power produced to the power available in the wind:

$$C_p = \frac{P_{mech}}{\frac{1}{2}\rho u_\infty^3 A} \tag{1.2.a}$$

This damage has not occurred for the duration of the CPWT's life. As a result, the present work does not further consider this type of blade damage. Long-term monitoring of the CPWT's power coefficient will also be complicated by new research showing significant deviations in C_p based on wind speed (see John Cunningham's thesis [10]),

Environmental patterns may also cause more significant damage to blades. Lightning strikes have been known to cause significant damage to blades – according to ref. [9], any wind turbine from the states of Texas, Kansas, and Illinois may expect blade damage from lightning strikes every 8.4 years. The most serious form of damage resulting from a lightning strike is tip detachment – or the removal of up to several meters of a large-scale turbine's blade. A photo of this is shown in Figure 1.1. Other damage sources include object impacts, such as hail, and ice accretion, where ice builds up on the blades of turbines in colder climates.

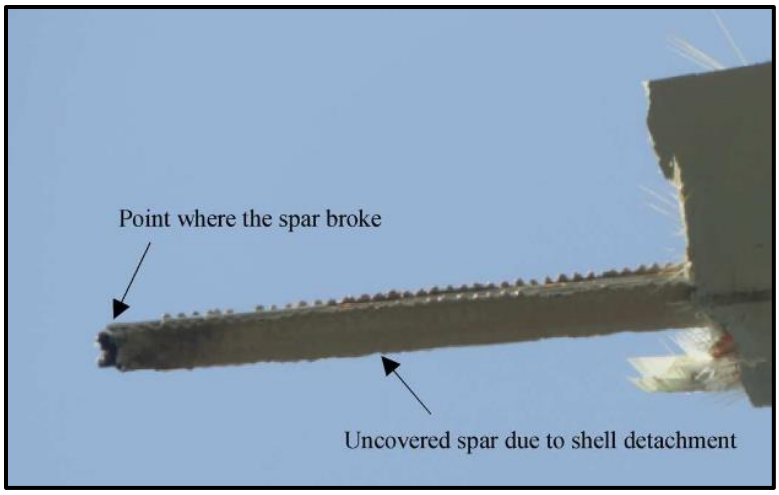


Figure 1.1: Photo of tip detachment due to lightning strike [9]

Environmental stresses may also cause finer damage to the composite structure of a blade. This may cause small delamination regions and cracks in the blade. Although these damages may not initially impact performance, the cyclic loading applied during operation will gradually cause damage growth. Eventually, this may lead to structural cracks, layer debonding, and large-scale buckling, deforming the blade structure overall. A more detailed account of structural damages may be found in ref. [8] and [9]. It is not enough to know why damage occurs, however. To adequately detect damage – especially without visual observation – requires knowledge of how the damage may functionally affect turbine operation.

Large-scale damage will cause significant changes to the nature of the forces acting on the blades. A significant loss of mass on a single blade, such as tip detachment, will cause the center of mass of the rotor to become offset from the shaft centerline. This results in a mass imbalance and causes an additional force to act transverse to the rotor. The magnitude of this additional force is given by the equation:

$$|\boldsymbol{F}| = mr\omega^2 \tag{1.2.b}$$

where **F** is a vector defined by the following diagram:

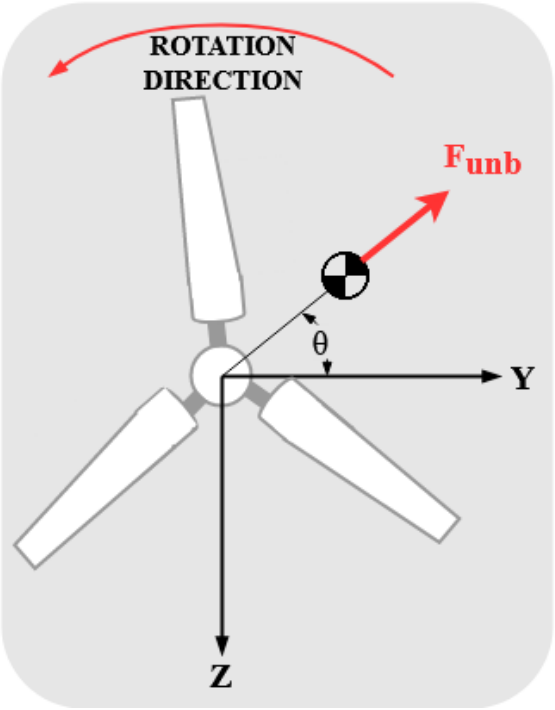


Figure 1.2: Force resulting from a mass imbalance. The shown coordinate system is based on the CPWT's accelerometer axes

The force **F** thus assumes a cyclic nature and can be given as a function of blade rotation θ according to:

$$\boldsymbol{F} = \begin{bmatrix} 0 \\ mr\omega^2 \cos\theta \\ -mr\omega^2 \sin\theta \end{bmatrix} \tag{1.2.c}$$

As shown by this equation, the imbalance force completes one cycle with every rotation of the turbine rotor – thus, it will affect the turbine at the frequency of the turbine rotational speed. This is known as the 1P frequency, as it occurs once per rotor rotation.

Damage that results in blade deformation will also cause significant changes to the aerodynamic properties of one or more blades. These aerodynamic changes result in variations of both axial and transverse forces. For a three-bladed turbine like the CPWT, this will cause vibrations in the rotor speed at the 1P and 2P frequency [11]. This can be visualized by considering the forces acting on blades only within a specific region; for example, consider only the forces

acting on a vertical upward blade. As blades pass through this region, a differential section of the blade develops an axial and transverse force component, as shown by Figure 1.3.

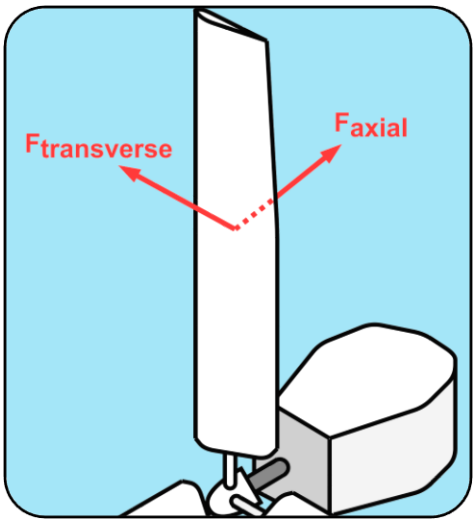


Figure 1.3: Diagram of axial and transverse forces experienced by a section of the turbine blade

The resultant force generated by a section of the turbine blade within this region is then given by:

$$F_{axial} = \frac{1}{2}\rho V_R^2 cdr(C_l \cos\phi + \mathrm{C_d} \sin\phi) \tag{1.2.d}$$

$$F_{transverse} = \frac{1}{2}\rho V_R^2 cdr(C_l \sin\phi - \mathrm{C_d} \cos\phi) \tag{1.2.e}$$

Where C_l and C_d are the lift and drag coefficients, respectively, and $\mathrm{V_R}$ and ϕ refer to the air velocity and direction relative to the airfoil. See ref. [12] for more detail on these equations. C_l and C_d are informed by the blade's airfoil profile. Blade damage causing an airfoil profile deformation will thus change these coefficients (likely decreasing C_l and increasing C_d). If only one blade is deformed, a 1P force will result from the decreased transverse force and increased axial force on the damaged blade occurring once per rotation, and a 2P force will result from the higher transverse force and lower axial force relative to the damaged blade occurring twice per rotation (owing to the two undamaged blades). This effect is demonstrated in ref. [11], which created a computer model of an aerodynamic imbalance.

Aerodynamic turbulence is also known to cause vibrations at the *n*P frequency, where *n* is multiples of the number of blades [12]. This effect is known as *rotationally sampled turbulence* and may be visualized by the diagram shown in Figure 1.4. As the blades "chop" through a turbulent vortex, each blade experiences forces at its rotation frequency. The combination of this effect on each blade of the turbine gives rise to *n*P vibrations; for the CPWT, this effect would occur at the 3P frequency.

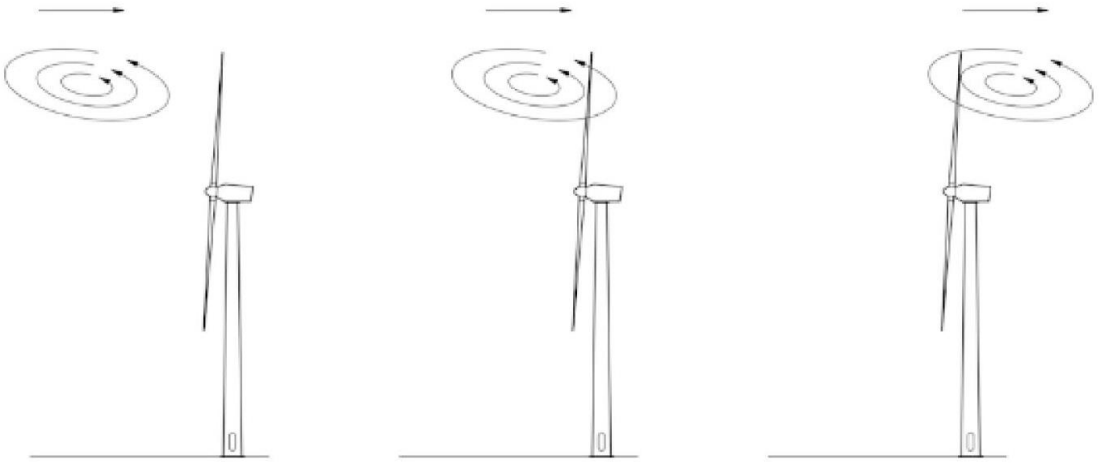


Figure 1.4: Rotationally sampled turbulence [12]

This effect is not likely to occur for the CPWT in a balanced state, as the rotor diameter (4m) is likely too small to experience large-scale turbulent vortices. Despite this, large damage may induce turbulent effects, and thus it is important to monitor for such an effect. Thus, for the CPWT, aerodynamic imbalances must be considered by monitoring the 1P, 2P, and 3P frequency components of tower vibrations.

Smaller damage may also occur, which are undetectable based on the theory described above. These require specialized methods for detection, which will be discussed at the end of the next section.

1.3 Previous Research

This section serves to review previous research and methods to detect damage to the blades of wind turbines. The turbines operate across several energy domains: they convert wind energy to mechanical energy and transmit this through the rotor to the generator, which outputs electrical energy. As a result, research on condition monitoring systems includes aerodynamic, rotor dynamic, and electrical analyses.

For a typical rotor dynamics approach, vibrational data is collected at critical locations of the turbine rotor. Vibrational measurement is also accompanied by other process sensors, including rotor speed and power output. This has been successfully applied by the National Renewable Energy Laboratory (NREL) for large-scale wind turbine gearboxes [13]. In these gearboxes, the multitude of bearings and gears necessitates the monitoring of multiple regions. An overview of the measurement locations can be found in Figure 1.5.

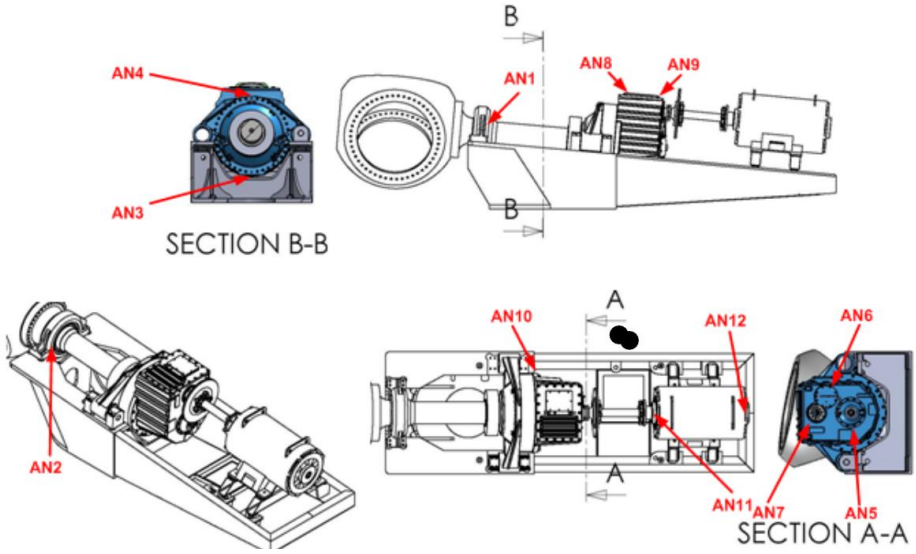
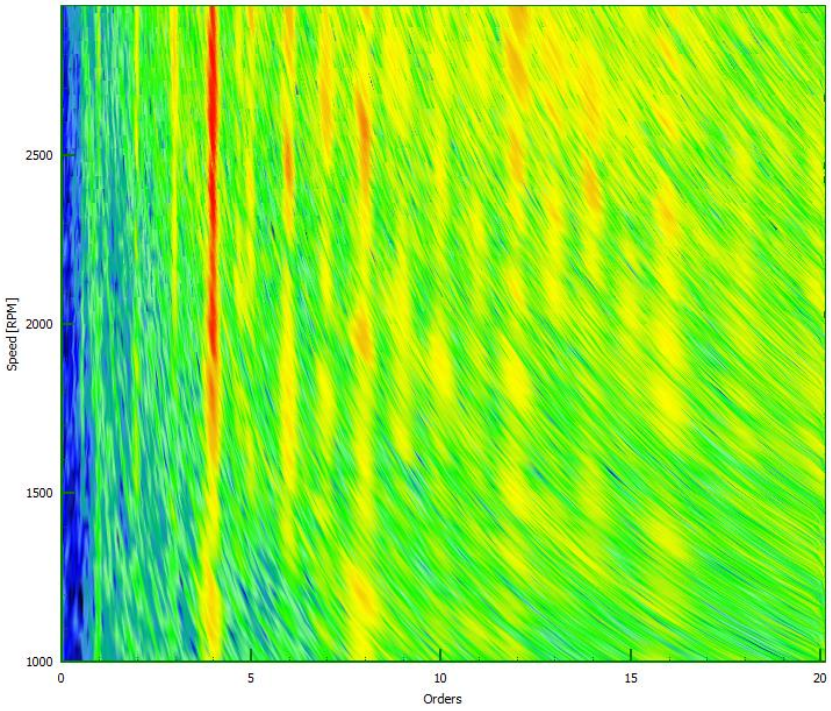


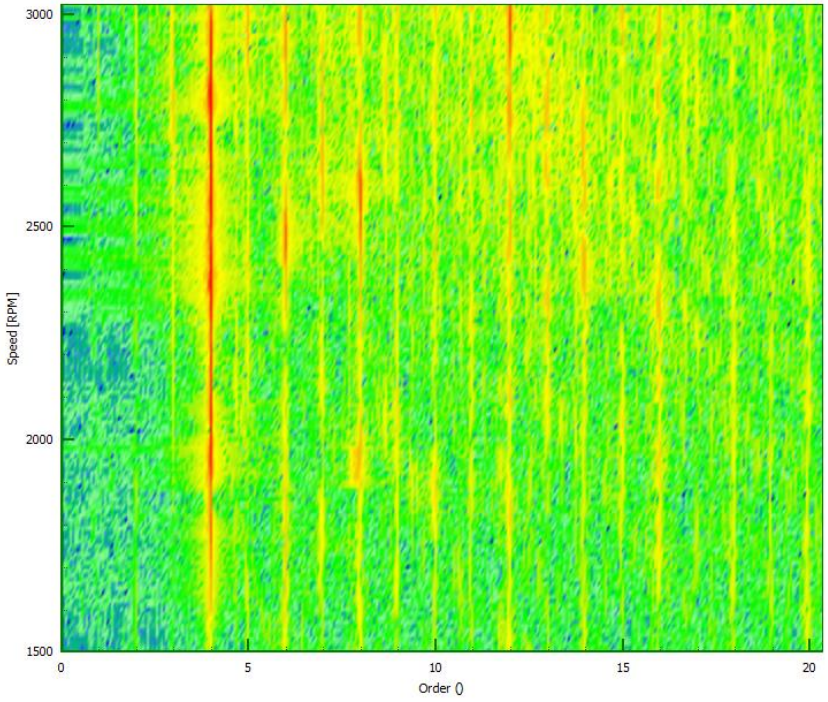
Figure 1.5: List of all accelerometers used to monitor NREL's 750kW GRC test turbine drivetrain [13]

Detecting faults using these accelerometers requires several data processing steps. The Fast Fourier Transform (FFT) is used to convert input data to the frequency domain; this allows for detecting faults appearing at specific frequencies. Before this is done, however, vibrational data is

processed using a technique known as *synchronous sampling*. Normally, acceleration data is collected in equal time increments. For variable speed devices such as wind turbines, however, significant changes in rotor speed over the duration of the signal to be transformed results in spectral power "smearing" across frequency bins. To fix this, acceleration data is resampled to increments in shaft rotation. The result of this may be seen in the waterfall plots of Figure 1.6, where synchronous sampling is applied to the vibration of a vehicle engine. Note that this is an extreme example, with rotor speeds 100-1000x that of a typical wind turbine. Another advantage of this method is that the Fast Fourier Transform frequency outputs in multiples of shaft rotating frequency rather than absolute frequency. This allows for better detection of faults occurring at multiples of shaft rotating frequency, like those 1P, 2P, and 3P effects described in section 1.2.



(a)



(b)

Figure 1.6: A waterfall plot of vibrational data processed in: (a) A constant timestep format (b) a synchronously sampled format [14]

The study in ref. [15] found that both mass imbalances and aerodynamic imbalances were detectable by measuring the rotor speed at 100 Hz. In this study, mass imbalances resulted in a clear excitation of the 1P frequency, which agrees with the theory presented in section 1.2. In detecting aerodynamic imbalances, their results agree with ref. [11], and showed that vibration peaks occurred at sidebands to the 3P frequency of 1P, 2P, 4P, and 5P. Their results are shown in Figure 1.7.

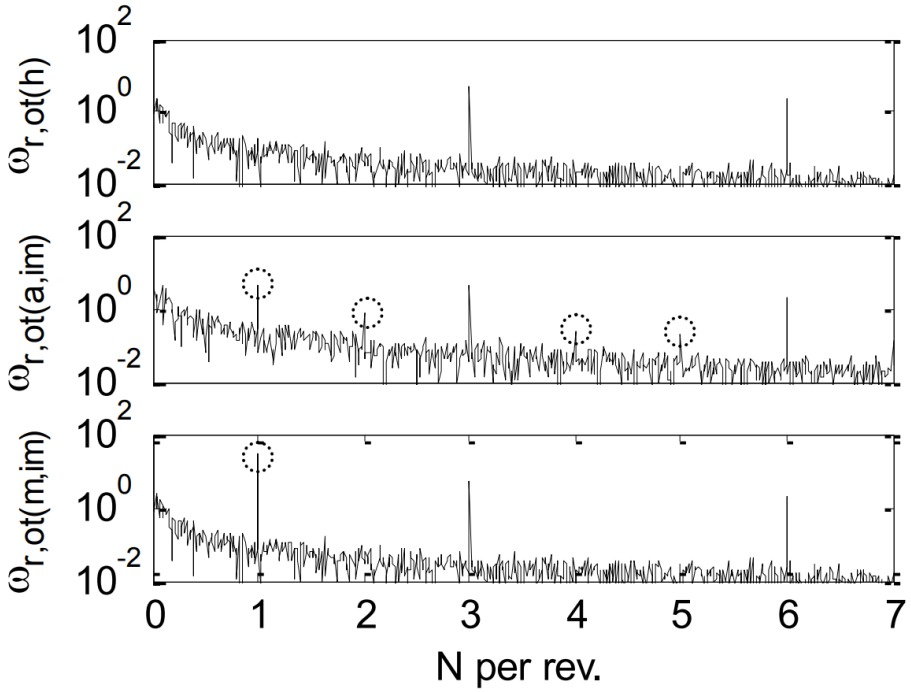


Figure 1.7: Frequency response of the synchronously sampled rotor speed signal. The top plot shows healthy operation, the middle plot shows operation with an aerodynamic imbalance, and the bottom plot shows operation with a mass imbalance. [15]

Another major approach to diagnosing blade damage is generator monitoring. The University of Nebraska has successfully identified mass damage in simulations by monitoring the electric current output from a small-scale direct-drive wind turbine [11]. They observed changes at the 1P frequency for mass imbalances, and changes at 1P and 2P for aerodynamic imbalances, again confirming the theory presented in section 1.2.

The previously discussed methods only detect blade damage after its magnitude significantly impacts turbine operation. Several specialized techniques exist to characterize damage by

monitoring the structure of the blades themselves. These techniques allow for the detection of blade damage much earlier than vibration sensors. Several types of solutions exist: strain measurements, acoustic emissions, and ultrasound sensors have all been used to characterize blade damage [17]. These structure-based monitoring systems are typically much more expensive than vibration monitoring equipment. As a result, they are less viable for small-scale distributed wind systems. Therefore, these methods will not be pursued further for the current version of LifeLine CMS.

After a review of the previously researched damage detection methods, and consideration of the availability of sensors already installed on the Cal Poly Wind Turbine, a rotor dynamics analysis focusing on accelerometer vibrational data was selected for continued study.

1.4 Project Objective

The mission of the present work is to design and implement a condition monitoring system (CMS) to detect blade damage. CMS design will be informed by real data collected from the CPWT; using this data, the CMS will be tuned so that it identifies faults and their related cause and minimizes the number of false positive detections. Once properly designed and tuned, the CMS will be implemented on the CPWT's control computer. Finally, the implemented system's performance will be validated through more testing. CMS development will follow the flowchart shown in Figure 1.8.

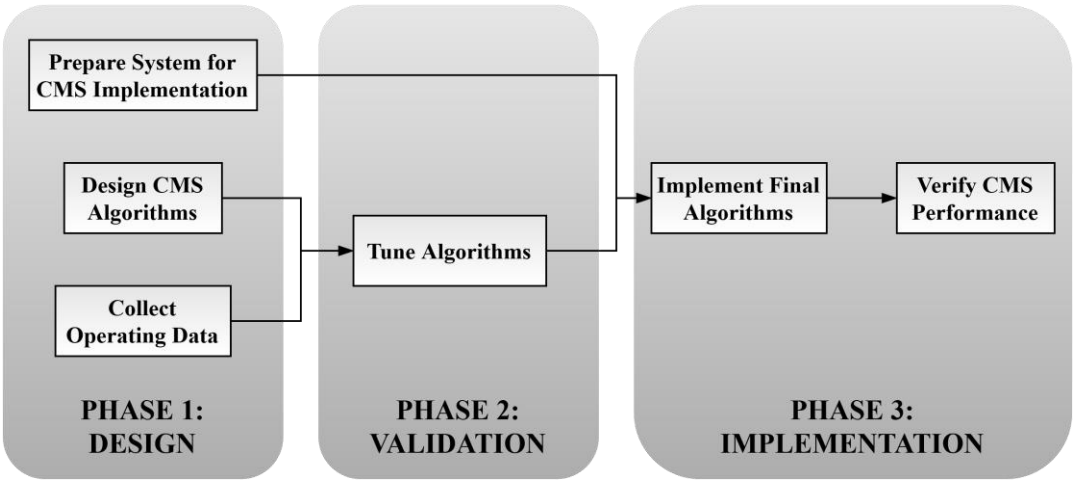


Figure 1.8: The three phases of CMS design and implementation

1.5 Thesis Layout

The rest of the present work is organized as follows. Chapter 2 details changes made to the Cal Poly Wind Turbine to improve the control system and data collection methods and allow for the installation of a fault detection software. Chapter 3 discusses the testing completed, intending to characterize the turbine's response to various operating parameters and to simulated blade damage. The rest of the paper is dedicated to the design and implementation of the CMS algorithms. Several fault detection developments are made in chapter 4, and their effectiveness is investigated in the cross-validation study of chapter 5. Finally, the on-site implementation is detailed in chapter 6. The present work concludes with suggestions for future steps in chapter 7.

Chapter 2

THE CAL POLY WIND TURBINE

This section serves as a short documentation of the Cal Poly Wind Turbine (CPWT) mechanical system, electrical system, and monitoring devices. This discussion informs algorithms to be deployed on the CPWT. Also discussed are changes made to the CPWT system throughout the present work. Two primary changes were implemented. First, integral control and preprogrammed test routines were added to the control system, which reduced steady-state control error and allowed for improved testing. Second, the electronic control and monitoring system was upgraded to aid live integration of fault detection algorithms; this change required a full rework of the electronic mounting hardware.

2.1 Turbine Overview and Tower Vibration Response

This section overviews the CPWT mechanical system, and details previous research to create theoretical models of this system. These theoretical models inform vibration-based fault detection algorithms and are validated in chapter 3. The CPWT is a small-scale, direct-drive horizontal-axis wind turbine developed by many student projects at California Polytechnic State University, San Luis Obispo, since 2008. Previous student works are summarized in Appendix C. Selection of Previous Student Works. The CPWT consists of a 70 3/8 ft steel tower supported by a gin pole, and a 12ft diameter turbine rated for a 3kW power output. This is shown in Figure 2.1. Power is dissipated via a resistive load into two water tanks at the turbine base.

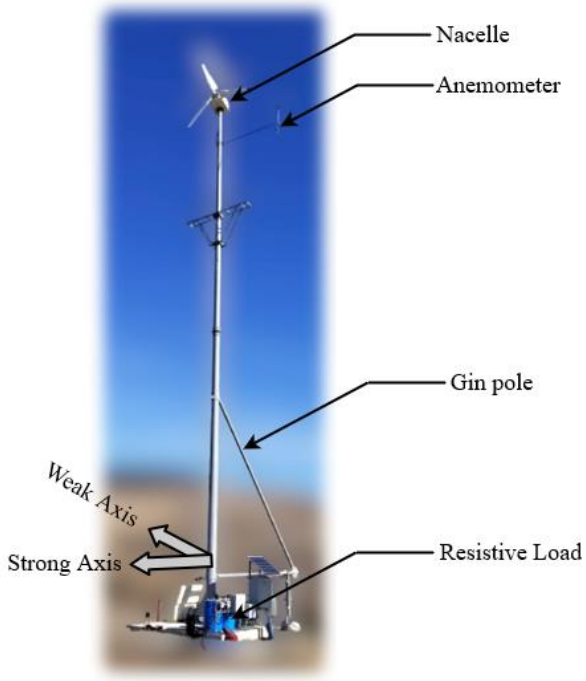


Figure 2.1: Photo of the CPWT

The gin pole supporting the tower results in the tower having a strong and weak axis, complicating the tower dynamic response. Several works have studied the theoretical response of the rotor/tower system. The rotor and tower vibration response are of particular importance, as the present work uses vibrational analysis as the main monitoring parameter for fault detection. Tom Gwon [18] and George Katsanis [19] both created theoretical tower models and calculated the resulting bending mode frequencies or resonant frequencies. Tower bending modes resulting from their analyses within the range of normal rotor speeds are shown in Table 2.1.

Table 2.1: Theoretical bending modes of the CPWT tower

Bending Mode	**Resonant Frequency [Hz]**	
	Katsanis	**Gwon**
1st Weak Axis	0.59	0.58
1st Strong Axis	0.81	0.83
2nd Weak Axis	2.97	2.81
2nd Strong Axis	4.70	4.96
3rd Weak Axis	8.05	8.13
3rd Strong Axis	12.99	13.90
4th Weak Axis	15.08	15.61

Furthermore, the natural frequency of the rotor blade theoretically influences nacelle vibrations. Katsanis used a simplified blade model and found that the natural frequency of the blade is 16.25Hz at rest. The blade experiences a centrifugal stiffening effect as the blade speed increases, according to the relationship:

$$\omega_R^2 = \omega_{NR}^2 + \alpha\Omega^2 \tag{2.1.a}$$

where ω_{NR} is the resting natural frequency, ω_R describes the rotating natural frequency at rotor speed Ω, and α is a parameter determined through additional analysis. Katsanis determined that $\alpha = 2.45\text{E-}3$.

Possible resonance occurring from a mass or aerodynamic imbalance is described by the Campbell diagram shown in Figure 2.2 and Figure 2.3. As discussed in section 1.2, a mass imbalance will result in vibration peaks occurring at the 1P frequency; thus, when the rotor speed is equal to a tower bending mode, tower vibrations will significantly increase. Likewise, an aerodynamic imbalance may result in 1P or 2P vibrations, causing a similar effect on tower vibrations if present. This increase may cause damage to the tower if vibrations are large enough. Also, a mass or aerodynamic imbalance inducing tower resonance may be an important metric for determining the presence of such an imbalance. These Campbell diagrams thus show where vibrational resonance may occur should damage be present, and also inform a wind turbine's control system – many wind turbine control systems seek to spend as little time in resonant regions as possible, to minimize the possibility of resonance-induced vibrations causing system damage.

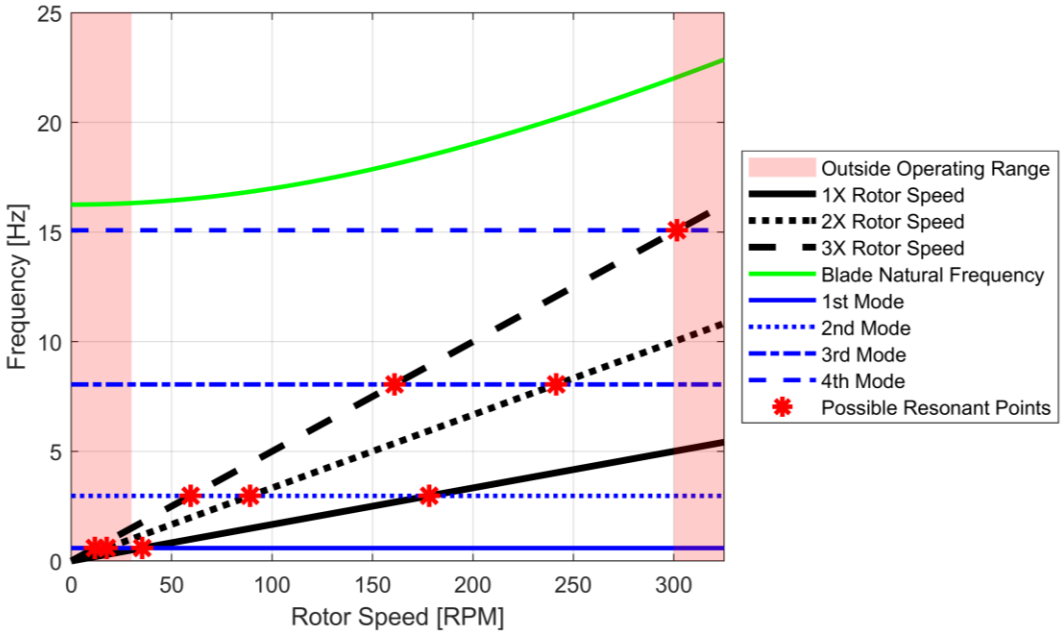


Figure 2.2: Campbell diagram for the tower weak axis

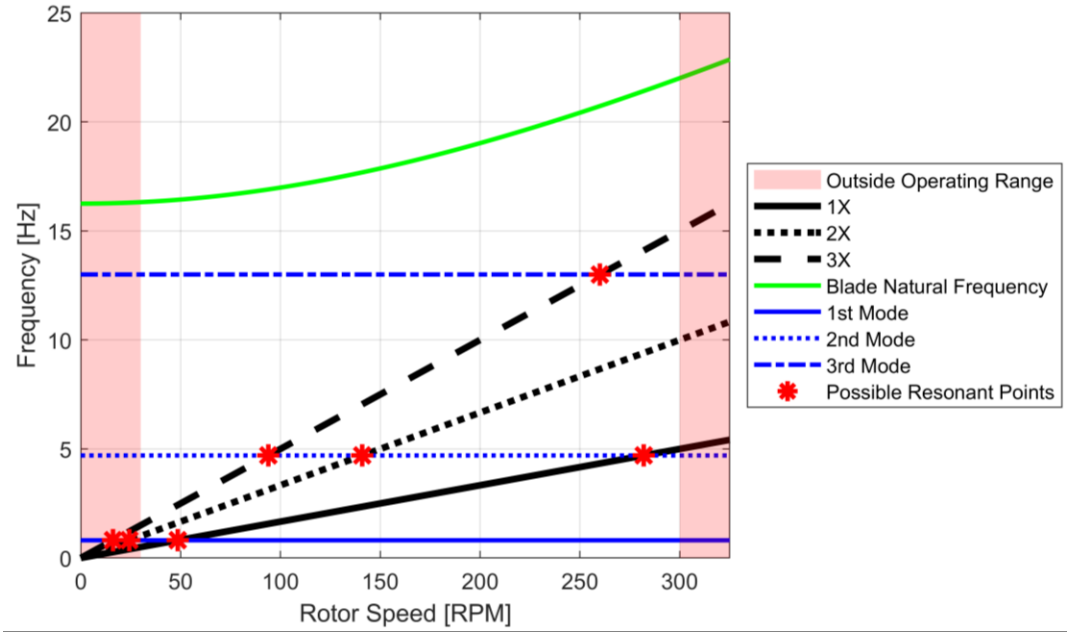


Figure 2.3: Campbell diagram for the tower strong axis

2.2 Control Method

The Cal Poly Wind Turbine (CPWT) is a small-scale wind turbine designed and manufactured at Cal Poly. Due to its small size and high rotor speed, a direct-drive transmission is used to generate

power using a GL-PMG-3500 permanent magnet generator. A diagram of the CPWT is shown in Figure 2.4 and includes a mechanical and electrical wiring diagram.

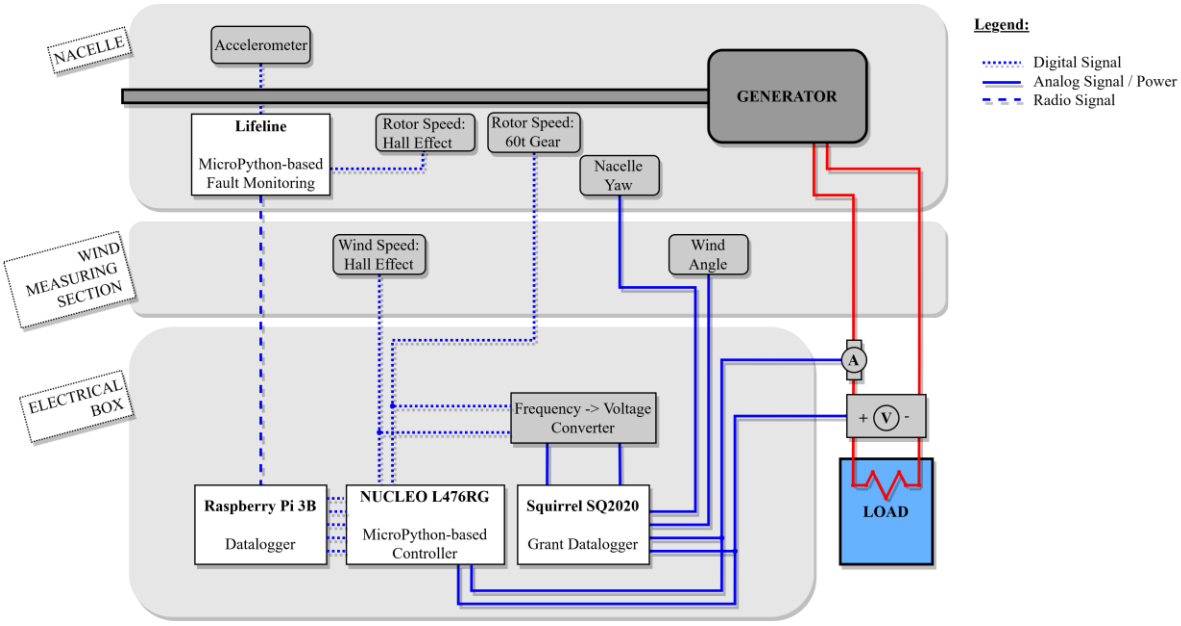


Figure 2.4: Overview of the CPWT electrical wiring, DAQ systems, and sensors.

The CPWT dissipates generator power into two large water tanks. The amount of power dissipated is controlled by a solid-state relay, which rapidly switches between closing and opening the loop. Controlling the duty cycle of the relay thus allows for controlling the rotor speed. The current control implementation is described by the block diagram shown in Figure 2.5. Note that the "Wind Turbine" transfer function is unknown; for efforts in creating a linear model, see Richard Sandret's thesis [39].

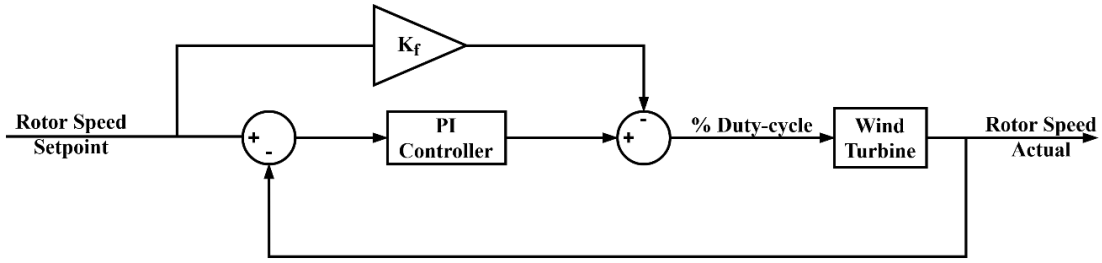


Figure 2.5: Rotor Speed Control Block Diagram. Note that the controller includes a feed-forward path, which estimates the desired duty cycle for any given rotor setpoint, and a typical proportional-integral controller.

At the beginning of the current project, the control method contained substantial steady-state error, as only proportional control was used. In typical operation, an error of up to 10RPM was typical. This limited the ability to control the rotor speed and impacted the precision of testing. Thus, integral control was added and tuned to minimize steady-state error. After testing and tuning, a proportional gain of 4, an integral gain of 0.015, and a feed-forward gain of 0.25 were selected. Gains were first selected based on Richard Sandret's work in creating a linear tower model [39]; however, these resulted in significant problems with overshoot and settling time. As such, the gains were empirically tuned to their final value. This reduced steady-state error to 0-2RPM in normal operation. It is likely that a refined linear model of the tower, once theoretically produced and empirically validated, could reduce this error even further.

However, the integral control created a new problem: integral windup. Integral windup occurs when the physical system cannot reach a desired setpoint, causing an accumulation of the integral term when calculating the setpoint. Should the emergency brake be pressed or the windspeed drop such that the rotor setpoint was unable to be reached, the integral error quickly accumulated to significant levels. In practice, such a windup led to a steady-state error of up to 20RPM. Thus, a simple anti-windup method was implemented that continually zeroed the integral error when the rotor speed error was larger than 10 RPM. This simple fix had the desired effect; outside the 10 RPM error region, the control system quickly adjusted towards the desired setpoint. Within this region, the control system was able to compensate for shifts in inputs, such as wind speed or yaw angle, reaching the desired value within several seconds.

Constant tip-speed ratio control, or TSR control, was also implemented. TSR control consists of holding the ratio between wind speed and the turbine blade tip-speed constant. TSR is defined by the following ratio:

$$\lambda = \frac{Blade\ Tip\ Speed}{Wind\ Speed} = \frac{\Omega R}{u_w} \tag{2.2.a}$$

The implemented TSR control is a modified version of the rotor speed control method, using the desired setpoint λ_{set} to calculate the desired rotor speed. The block diagram for this control system is shown in Figure 2.6.

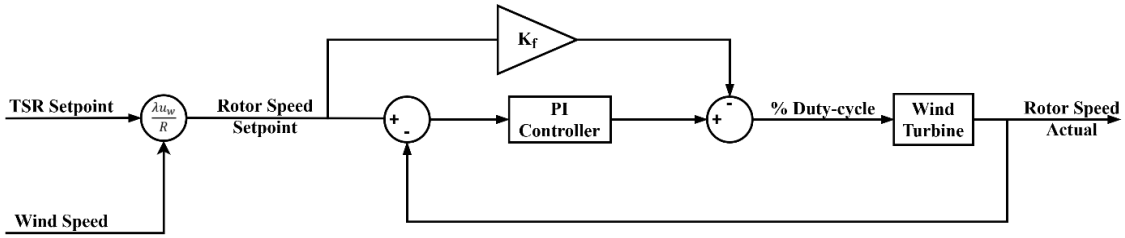


Figure 2.6: TSR Control Block Diagram

Variations in wind speed resulted in large instantaneous changes of λ, causing the first version of TSR control to rapidly change the duty cycle – faster than the system could respond. To remedy this, a low-pass filter was applied to windspeed readings used to calculate λ. For timesteps n and $n+1$, the low-pass filter was used:

$$u_{w,filtered}^{n+1} = A * u_{w,filtered}^{n} + B * u_{w,raw}^{n+1} \quad (2.2.b)$$

This filter had the effect of smoothing out large changes in wind turbine readings, giving the CPWT control system more time to adjust to changes in wind speed – and the ability to ignore short-term spikes in wind speed. The values A = 0.95 and B = 0.05 were chosen via trial and error, balancing between unfiltered data (resulting in the duty cycle changing faster than the turbine can accommodate) and over-filtered data (causing a large λ error). This filter was only applied to wind speed used to calculate λ - not the recorded wind speed.

The implemented control method is unable to perfectly react to changes in the operating state, especially large yaw angle and wind speed changes. These may likely be reduced through the implementation of a more complex control method; however, these errors did not impact testing results and as such were not pursued.

The final change made to the control system was the introduction of automated tests. The Raspberry Pi runs a GUI that allows the user to manually change the control setpoint and choose between constant rotor speed and constant TSR control. The added code allows the user to start premade tests. These tests are created via a .csv file. This allows for finely executed tests like those described in chapter 3.

2.3 Data-collection System

This section details the equipment used for data acquisition on the wind turbine and describes several improvements made to the equipment throughout the present work. The CPWT possesses three independent data-collection systems. The CPWT control system uses two sensors – rotor speed and wind speed – and reports control parameters such as setpoints and the control duty cycle. The LifeLine independently records three-dimensional nacelle vibrations. Additional data is also collected by the Grant SQ2020 (Squirrel) system. Table 2.2 lists all parameters recorded by the system. LifeLine parameters marked with an asterisk (*) correspond to sensors added as a part of Ryan Zhan's thesis [6].

Table 2.2: Parameters recorded by each DAQ system

System	Parameters Measured	Collection Rate [Hz]
Controller	Rotor Speed [RPM] Wind Speed [m/s] Duty Cycle [%]	5
Squirrel	Rotor Speed [RPM] Wind Speed [m/s] Generator Current [A_{DC}] Generator Voltage [V_{DC}] Wind Direction [deg] Nacelle Yaw [deg]	1
LifeLine	Nacelle Acceleration [g*16384] Rotor Speed [RPM]* Generator Current [A_{DC}]* Generator Voltage [V_{DC}]*	50 1 50 50

Various factors offset the time of data recording. The Raspberry Pi (RPi) suffered several delays in collecting data from the controller and LifeLine. Serial communication delays occurred when the RPi did not immediately record data sent by the controller or LifeLine. Radio

communication also delayed the LifeLine data even further. Finally, the internal clock of the RPi stopped when powered off. This required the manual synchronization of the Squirrel and RPi clocks, which was very difficult to do perfectly. As a result, analyzing data required careful alignment of the datasets by time. See chapter 2 for further discussion of the alignment process. Misalignments of up to 20 seconds between the LifeLine and Squirrel were common. To remedy the alignment issues described above, sensors recorded by the Squirrel were also added to the controller. By doing this, the Raspberry Pi became the central device recording all data necessary for a fault detection algorithm to function. This setup minimized possible alignment errors and reduced the number of devices necessary for proper data-collection to two. Unfortunately, these improvements were not completed in time for the testing described in chapter 3; however, future testing can make use of these. Future work might also add an external clock to synchronize each device's internal clock – although communication with the Squirrel SQ2020 (a closed-environment system) may prove difficult.

To facilitate adding the sensors from the Squirrel to the controller, the expansion board for the controller was redesigned by Dr. John Ridgely. In redesigning this board, external wiring was condensed onto the board and additional analog pins were added. This redesign required that the electronics mounting unit in the electrical box be updated with a new version. The new version mounted the RPi monitor, terminal block, and buck converter to an aluminum plate. The RPi and controller were mounted on an acrylic sheet fixed to the aluminum plate via metal standoffs. This allowed for the RPi and controller to be removed and serviced without requiring the removal of the entire plate from the DIN rails it was mounted on. The old and newly updated electronics mounting unit are shown in Figure 2.7 and Figure 2.8, respectively.

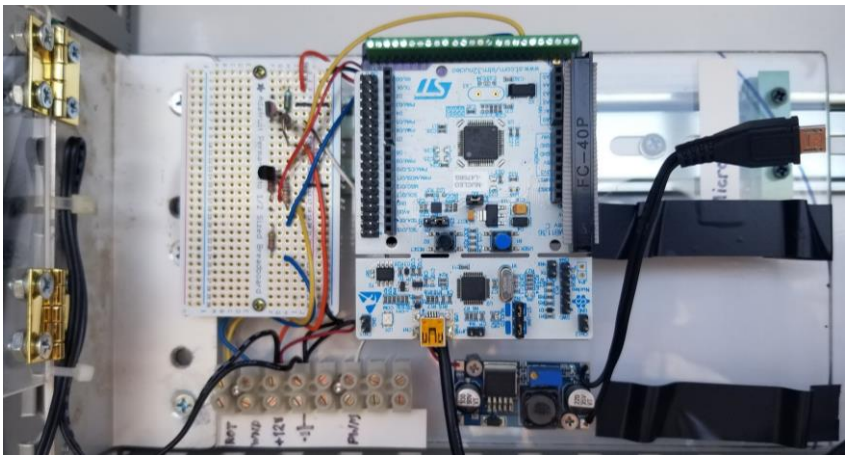


Figure 2.7: Old electronics mounting unit for the RPi and CPWT controller. The RPi is not pictured.

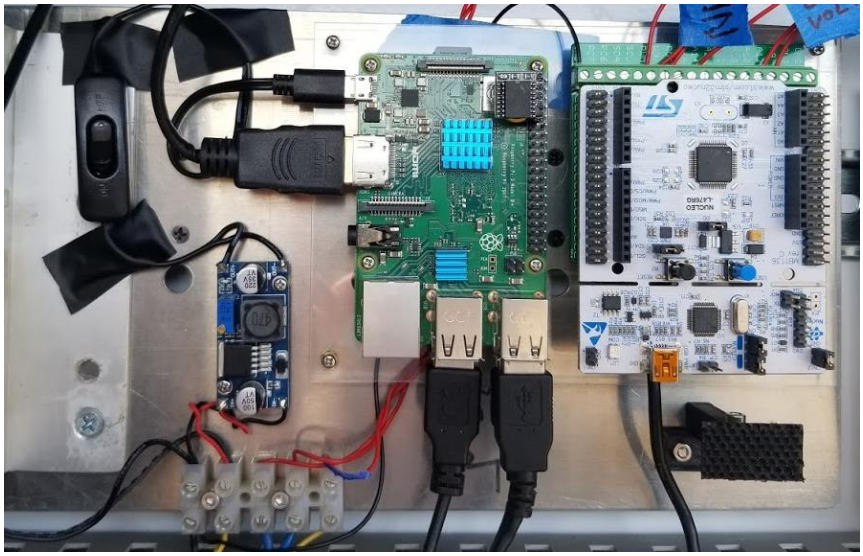


Figure 2.8: Electronics mounting unit for the RPi and CPWT controller

To conclude, several changes were made to the CPWT to prepare the system for installation of a condition monitoring system. The control system was improved by adding integral control and Tip-Speed Ratio control. This, along with several smaller fixes, reduced steady-state error from ~10RPM to ~2RPM. An automated testing routing was also introduced, allowing for the preprogramming and automatic execution of tests. Finally, a new electronic mounting unit was built, allowing for the addition of new sensors, and simplifying the installation of a condition monitoring system. This paved the way for field testing to characterize the vibrational response of the turbine to rotor speed changes, wind speed changes, and simulated damage – a topic that will be discussed in chapter 3.

Chapter 3

PHYSICAL TESTING

This thesis takes a data-driven approach to fault detection. Any algorithm developed must be both trained by and tested on operational data. Fault detection focuses specifically on large-scale damage to turbine blades that would cause a significant portion of the blade to be lost. Testing must be completed in a repeatable way, to control for any factor that might affect the tower vibrational response. In other words, each test must be conducted in a near-identical fashion. Also, the environmental conditions acting on the turbine must be as similar as possible. Blade damage is simulated by attaching a premeasured imbalance mass to the blade. As nacelle vibrations are the major fault detection parameter, the testing goal is to minimize any potential factor that might impact nacelle vibrations *besides* an added mass.

The other major focus of this chapter is to identify features, or methods of post-processing operating data, that best identify imbalance faults. In general, post-processing of any data takes two forms: time-series and frequency-based methods. For time-series methods, any feature must account for the fact that wind (and thus the vibration response) is an inherently stochastic process. To account for this, many sources including ref. [20] recommend using statistical processing methods. The Fast Fourier transform (FFT) will be used for frequency analysis.

3.1 Experimental Design and Data Collection

The main goal of testing was to collect a variety of system operating data in a healthy and faulty operating state in a repeatable fashion. To accomplish this, two major types of tests were completed: steady-state tests and ramp tests.

For steady-state tests, the rotor speed was held constant in increments of 20RPM, from 80 to 180RPM. At each increment, five minutes of data were collected. These tests were undertaken to

better understand the behavior of the rotor while holding the rotor speed constant. In particular, the effect of varying wind-speed on tower vibrations was investigated. Thus, six sets of data were collected for each rotor configuration.

For ramp tests, the rotor speed was incremented by 1 RPM every 10 seconds, from 30 to 210 RPM. Each test therefore collected 30 minutes of data. These tests were completed for two purposes. First, the ramp test allowed for analyzing the frequency spectra of the vibration signal as a function of the rotor speed. Specifically, this test allowed for verifying the theory presented in section 1.2, which states that mass imbalances appear at the 1P frequency and aerodynamic imbalances appear at the 1P and 2P frequencies. It also allowed for verifying the tower bending modes as described by the Campbell diagram in Figure 2.2. Second, the cross-validation study of chapter 5 requires the division of all healthy data into ten sets, with each run including the full operating range of the system; therefore, each ramp test will correspond to one set of data for this study.

For the cross-validation study, data was only collected when the wind adhered to two characteristics:

1. The wind speed must be relatively similar and uniform for each test.
2. Wind direction must be predominantly westerly, to reduce tower strong-weak axis interactions

The first criterion allows for the most accurate comparison between tests. The second criterion reduces tower strong-weak axis interactions; see chapter 2 for more information. The west-to-east wind direction was ideal, as it is the most common direction at the turbine site in absence of large-scale weather patterns.

Both the steady-state and ramp tests were completed with the rotor in a balanced “healthy” state, and an intentionally imbalanced “faulty” state. For the faulty state, a sheet metal mass was duct-taped to the end of the wind turbine blade. Three unbalance masses were tested: 50g, 100g, and 200g. The exact mass of the sheet metal and duct tape combinations are listed in Table 3.1.

Table 3.1: List of masses used in testing, along with the exact weight added including duct tape

Nominal Mass [g]	Total Mass [g]
50g	64
100g	119.2
200g	223.5

Figure 3.1 shows the 100g mass attached to the turbine blade. The 100g and 50g imbalance were attached to the "suction" side of the blade, or the downwind side. The 200g imbalance was attached to the "pressure" side of the blade, or the side facing the wind. It is important to note that this simulated faulty state is not dynamically identical to true blade damage; with a large section of blade missing, the lift produced by that blade will significantly decrease. This reduction in lift will produce a cyclic load on the nacelle at 1X and 2X the turbine rotating frequency. The attached mass may reduce the produced lift, but not to the same extent as the damage described above. Furthermore, the added mass produces a dynamic effect opposite to that of a damaged blade: rather than offsetting the rotor center of mass *away* from the damage location, the added mass shifts the center of mass *toward* the location of simulated damage. Although the dynamics might vary from the true damaged state, the simulated damage is assumed similar enough to still provide meaningful insight towards how to best detect damage.

Figure 3.1: Test setup of 100g mass imbalance. The mass is attached to the downwind, "suction" side of the blade.

As discussed in chapter 2, the CPWT has three independent data collection systems: the LifeLine, which collects tower vibrations, the Nucleo controller, which collects useful control information, and the Squirrel SQ2020, which monitors all other system parameters. See Table 2.2 for more information. Each of these systems must be started separately. Additionally, at the beginning of data collection, only the LifeLine and the Squirrel SQ2020 recorded all data necessary for the fault detection algorithms used in this thesis; as a result, these systems were the only ones used for post-processing and analysis. Processing of data followed two major pathways: for analysis of the turbine vibration response, and integration into each fault-detection algorithm. This processing flowchart is detailed in Figure 3.2. During preprocessing, data is saved to the system memory as an intermediate step so that time spent processing for plotting and fault detection is reduced.

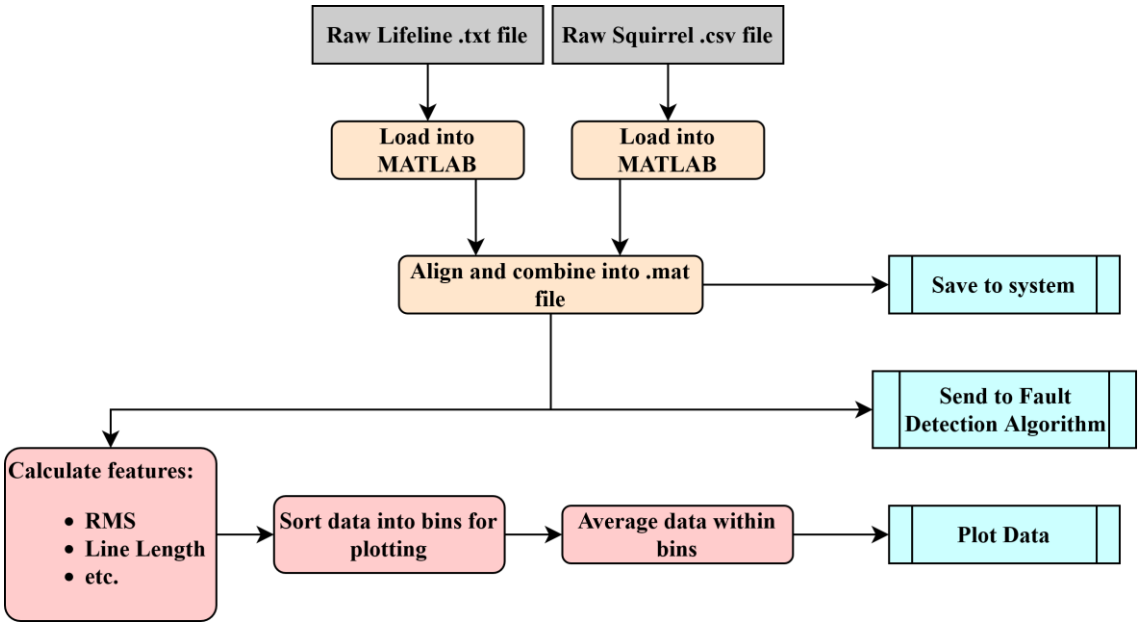


Figure 3.2: Processing flowchart to prepare data for analysis and fault detection

As the LifeLine initially only recorded vibrational data, testing necessitated additional steps to generate “markers” that could be used to align vibrational data and Squirrel data. Ultimately, the chosen marker was to bring the turbine to 210RPM, then set the duty cycle to 100%. This quickly brought the turbine to a stop and generated a sharp peak in vibrations. Creating two of these markers

generated enough information to properly align the two sets of data; this process is shown in Figure 3.3.

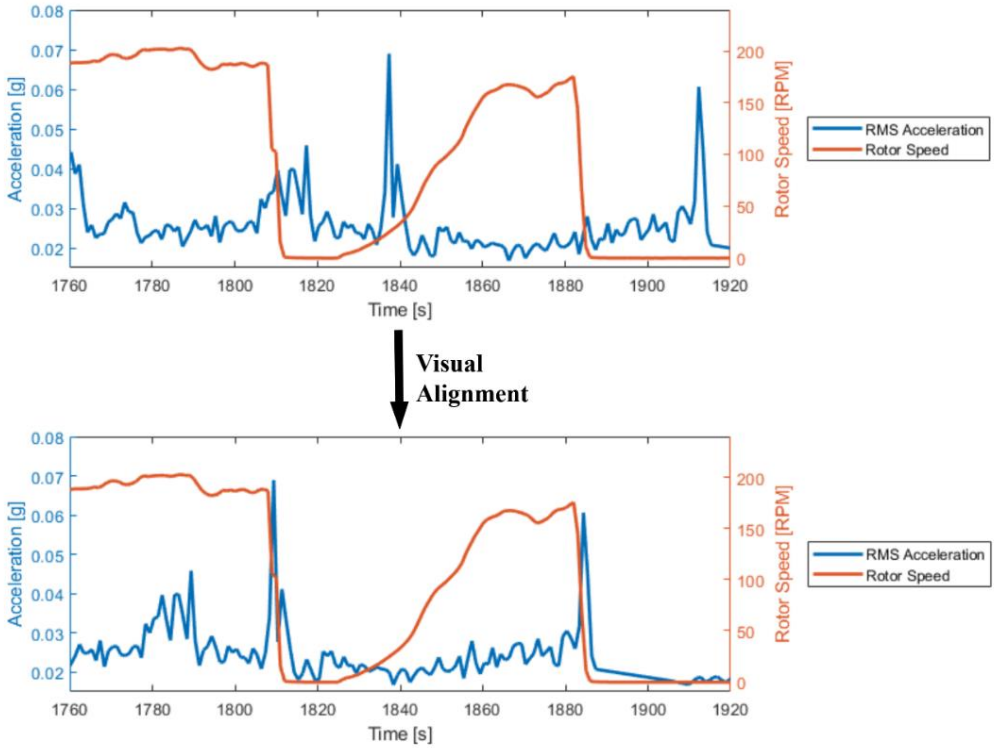


Figure 3.3: Alignment process using vibration markers from quickly stopping the turbine.

The addition of a rotor speed sensor to the LifeLine, as discussed in chapter 2.3, allowed for verifying the accuracy of this method. As shown in Figure 3.4, the peak in vibration closely followed the drop in rotor speed. Therefore, alignment error can be kept under three seconds.

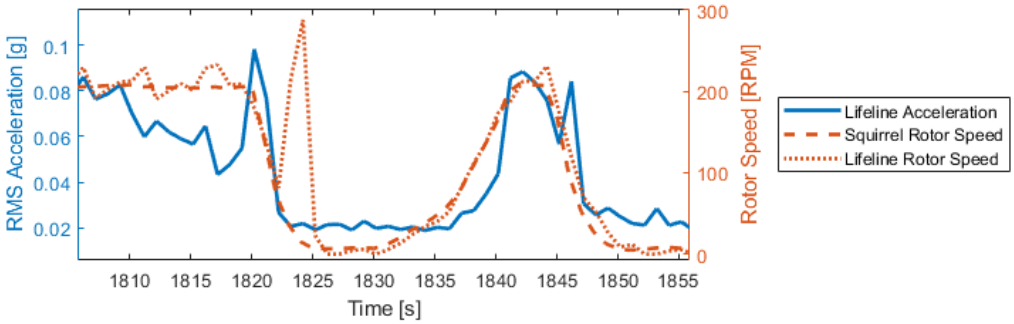


Figure 3.4: Aligned data with the rotor speed sensor installed on the LifeLine. The spike in rotor speed from the LifeLine at 1824 seconds is an incorrect data point related to noise.

The wireless transmission of data between the LifeLine and Raspberry Pi also sometimes ended in a significant amount of data being lost. This was typically caused by the positioning of the

antenna receiving data. Thus, it was necessary to promptly post-process the data to ensure minimal data was lost. Roughly 25% of tests did not produce meaningful results due to this issue.

3.2 Response Characteristics

Several features are theoretically useful for defining the vibrational response of the nacelle. This section defines these features. For the time-series analysis of vibrations, statistical analysis is commonly used. This is due to the stochastic nature of wind, which makes describing a vibration signal with a direct formula very difficult [20]. Therefore, several statistical features are defined here. All measures are computed for each axis of tower vibrations as recorded by the MMA8452Q accelerometer.

The simplest of these measures is the root-mean-square value, which is used to describe the average magnitude of vibrations. It is defined by the equation:

$$x_{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} \tag{3.2.a}$$

The RMS value is used for two reasons. First, it is a better measure of average vibration amplitude than the true average, which is very often zero (and thus unable to be used by the algorithm). Second, it is resistant to random peaks in a steady-state operating condition [20].

Another major parameter used is the Line Length of the signal, a health prognostics tool used to detect damage to helicopter blades [16]. Computationally, it is the sum of the distance between all data points, where each data point is considered a two-dimensional point $\boldsymbol{x} = (t, x)$. Here, *t* is the timestamp of the measured data point, and *x* is the sensor value.

$$x_{LL} = \sum_{i}^{n-1} |\boldsymbol{x}_{i+1} - \boldsymbol{x}_i|^{0.5} \tag{3.2.b}$$

Next, the Crest Factor is the ratio between the peak amplitude of the signal and the RMS value:

$$x_{CF} = \frac{x_p}{x_{RMS}} \tag{3.2.c}$$

Ref. [21] showed that the crest factor indicates unbalance in an electric motor, hence its inclusion here.

The Shape Factor is another indicator useful in characterizing unbalance faults, according to ref. [20]. It is the ratio between the RMS value and the average absolute value of the signal, squared:

$$x_{SF} = \frac{x_{RMS}}{\left(\frac{1}{n}\sum_i^n |x_i|\right)^2} \tag{3.2.d}$$

The final common statistical feature used is Kurtosis, which has also been shown to signify mass unbalance in electric motors by ref. [21].

$$x_{KURT} = \frac{\sum_{i=1}^n (x_i - \bar{x})^4}{n\sigma_x^4} \tag{3.2.e}$$

Frequency-based analyses also often show fault conditions. Mass unbalance, for example, appears as a peak at 1X the operating speed of rotor-based machinery, and aerodynamic imbalances often occur at 1X and 2X the operating speed. The discrete Fourier transform, or DFT, decomposes a signal into the frequencies that compose it and allows for identifying these frequency peaks. The Fast Fourier Transform, or FFT, computes almost the same result as the DFT - but requires significantly less processing time. This thesis uses the FFT due to the large datasets studied. The FFT takes a signal of length *n* and returns a matrix of complex vectors, also of length *n*. Each vector corresponds to a specific frequency of vibrations. The absolute value of each vector then describes the amplitude of vibrations at the associated frequency.

3.3 Results

The LifeLine accelerometer reports acceleration in three axes. These axes are shown in Figure 3.5. As a potential mass imbalance theoretically adds a forcing frequency to the transverse (Y) axis, signal analysis will be done using data from this axis unless otherwise noted. The rotor faced

approximately west for all tests, aligning the Y-axis with the weak axis. As a result, the expected bending modes for the Y-axis vibrations are described by the weak-axis Campbell diagram shown in Figure 2.2.

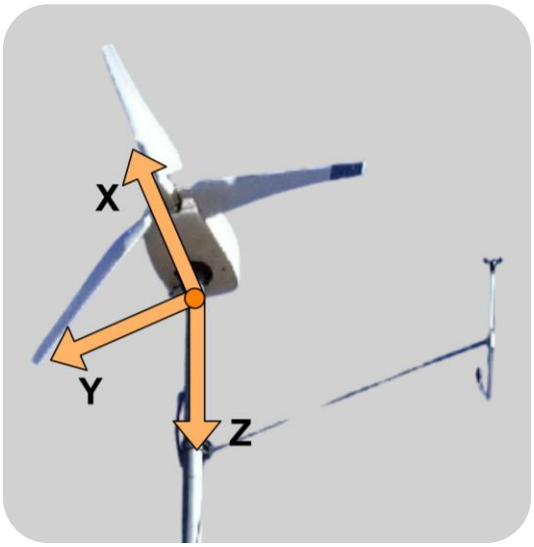


Figure 3.5: Accelerometer axes. The X-axis is in-line with the rotor, the Y-axis is in the transverse horizontal direction, and the Z-axis points down the tower.

For the balanced rotor, the RMS vibrations steadily increase up to 65 RPM, then stay relatively constant. The unbalanced rotor is similar, except that the RMS vibrations are higher and peak at 160 RPM. These vibrations could be visually seen during testing, as the tower began noticeably shaking. This 160RPM peak approximately corresponds to the 2nd bending mode of the weak axis, which has a natural frequency of 2.8 Hz (Gwon, [18]) or 2.97Hz (Katsanis, [19]) – corresponding to a rotor speed of 168 – 180RPM. This confirms the theoretical models described by the Campbell diagram in Figure 2.2, as this peak corresponds to the intersection of the 1P frequency and the 2nd bending mode. This result slightly disagrees with Derek Simon's work in ref. [22], which showed a constant RMS acceleration for a balanced rotor, and a positive correlation between these variables for an unbalanced rotor – without the associated peak at 160RPM. Since his work, the CPWT has been upgraded from a passive yaw system to an electromechanical auto-yaw system. As a result, there is always a slight error between the yaw angle and the incoming wind direction. This causes additional dynamic effects in the interaction between the incoming wind vector, the blades, and the

nacelle [12]. This is a likely explanation for variations in RMS accelerations with changes in rotor speed. The acceleration intensity also increases with wind speed, as shown in Figure 3.7; the effect is more pronounced for the imbalanced rotor.

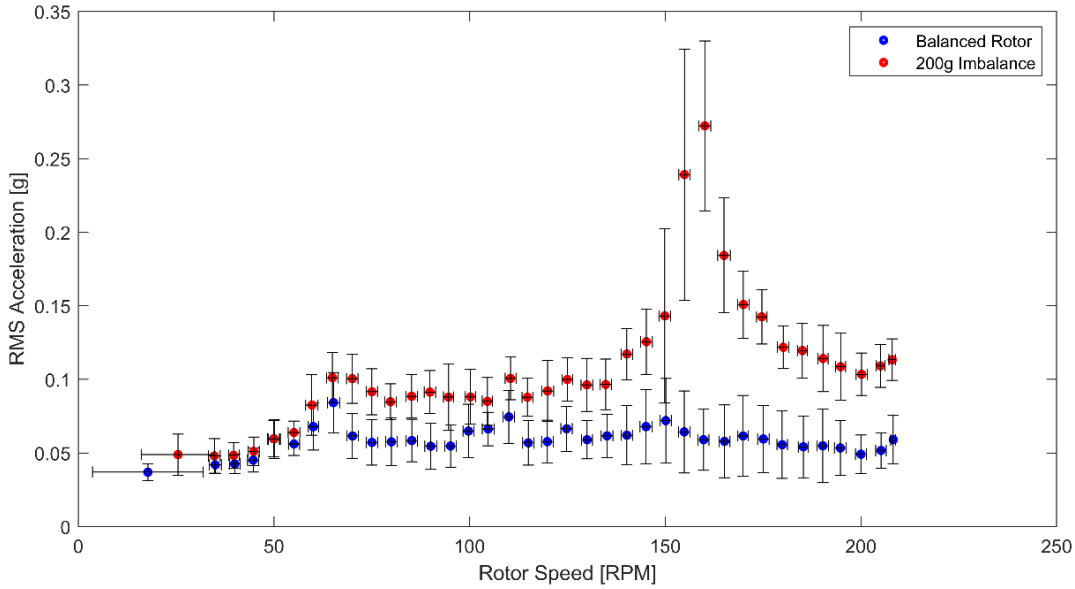


Figure 3.6: Plot of RMS Acceleration vs Rotor Speed

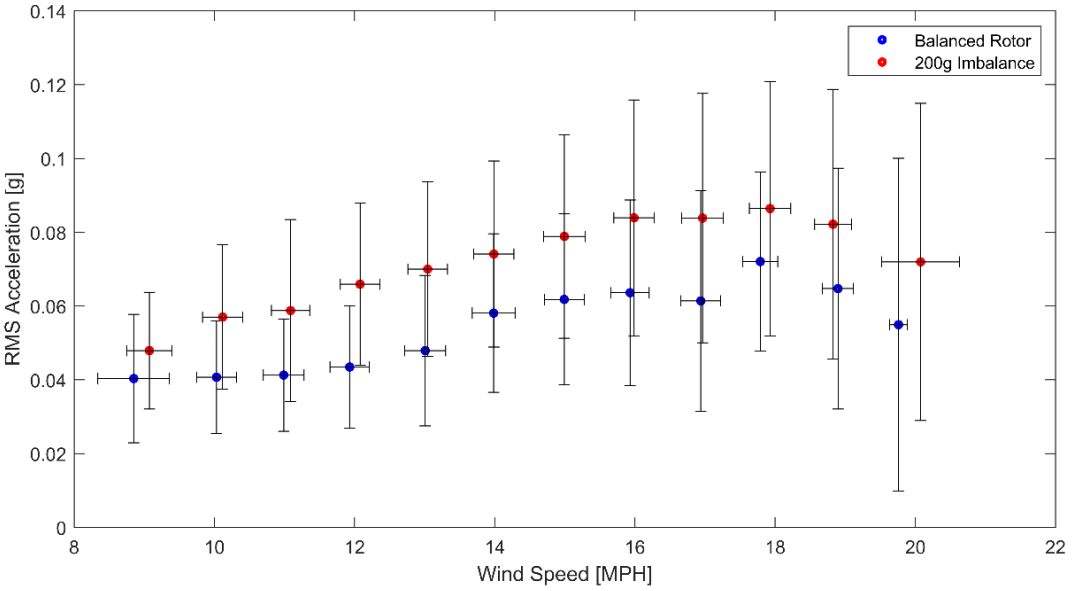


Figure 3.7: Plot of RMS Acceleration vs Wind Speed across all rotor speeds

The main use of the other statistical parameters detailed in section 3.2 is in differentiating between a healthy rotor and an imbalanced rotor. Table 3.2 shows the average percent difference between the healthy state and all masses tested. As shown, the Line Length measure most clearly

differentiates between a balanced and unbalanced rotor: with a 200g imbalance, the Y-axis Line Length measure has a 111% change from the healthy value. The RMS measure is the second-best predictor of faults, with an 82.6% change from the healthy value to the 200g imbalanced value.

Table 3.2: Average values for parameters for ramp tests and percent change from a balanced rotor

Parameter	Axis	Average Value				Percent Change from Healthy		
		Healthy	50g	100g	200g	50g	100g	200g
RMS	X	0.0467	0.0706	0.0592	0.0739	51.3	26.7	58.4
	Y	0.0586	0.093	0.0851	0.1070	59.2	45.2	82.6
	Z	1.0054	1.0073	1.0070	1.0072	0.2	0.2	0.2
Line Length	X	34729	52008	47001	57967	49.8	35.3	66.9
	Y	31529	53531	50549	66560	69.5	60.3	111.1
	Z	37865	59142	56477	65120	56.2	49.2	72
Crest Factor	X	2.4507	2.5216	2.5619	2.5750	2.9	4.5	5.1
	Y	2.4408	2.5456	2.4732	2.4505	4.3	1.3	0.4
	Z	1.1096	1.1746	1.1546	1.1920	5.9	4.1	7.4
Shape Factor	X	0.0024	0.0017	0.0020	0.0016	-29.7	-14.0	-32.5
	Y	0.0017	0.0012	0.0013	0.0010	-28.0	-24.1	-37.3
	Z	0.0001	0.0001	0.0001	0.0001	0.2	0.1	0.4
Kurtosis	X	3.0441	3.0295	3.0469	3.1258	-0.5	0.1	2.7
	Y	3.4084	3.1495	3.0390	2.9018	-7.6	-10.8	-14.9
	Z	3.1772	2.9962	3.0143	2.9562	-5.7	-5.1	-7.0

For frequency-based analysis, a single FFT reveals some information about the rotor operating state. However, a single FFT is unable to describe the effect of varying rotor speed. This is better characterized by creating a waterfall plot, which shows how the frequency spectra change with rotor speed. To create these plots, every 1024 data points of the vibration signal are processed via an FFT. The average rotor speed for each FFT is computed. The FFT outputs are then organized into bins of 5RPM increments, from 30 to 210RPM. Finally, all FFTs within a single bin are averaged to form a single frequency spectrum. Once the FFT outputs from each bin are computed, they are assembled into a 3D mesh plot. The results of this process are shown in Figure 3.8, Figure 3.9, and Figure 3.10 (corresponding to a balance rotor, a 100g imbalance, and a 200g imbalance, respectively).

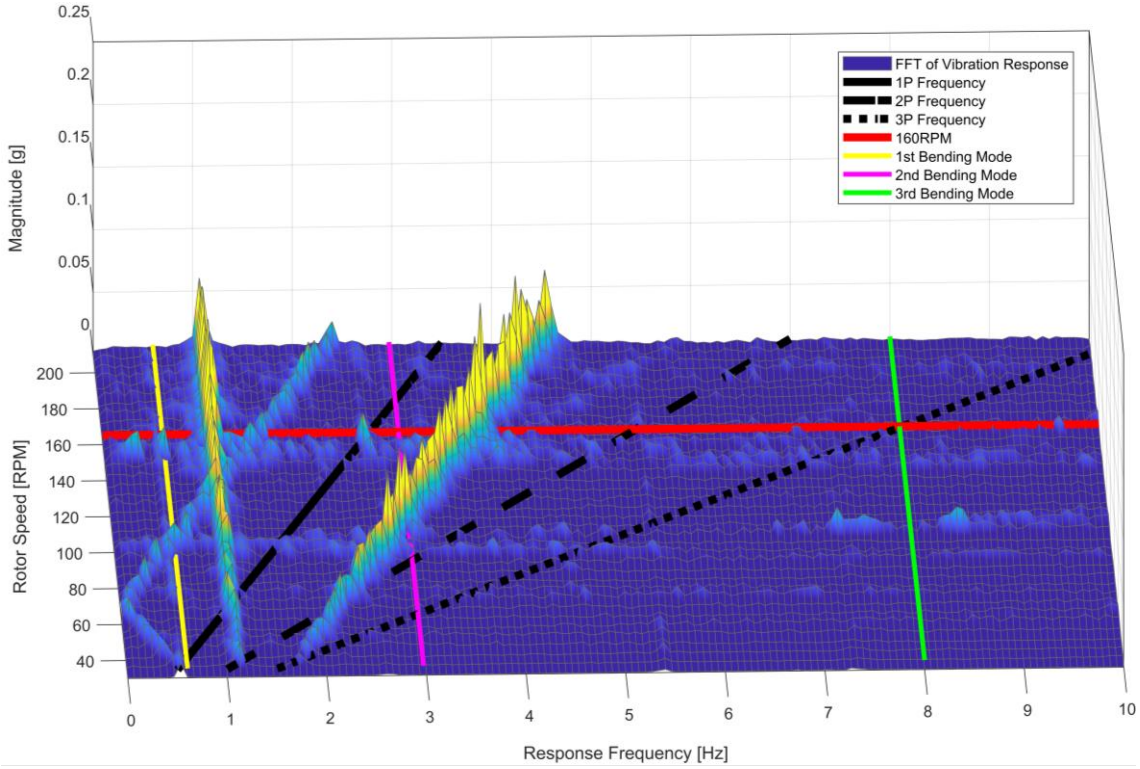


Figure 3.8: Waterfall plot of frequency spectra as rotor speed changes with a balanced rotor

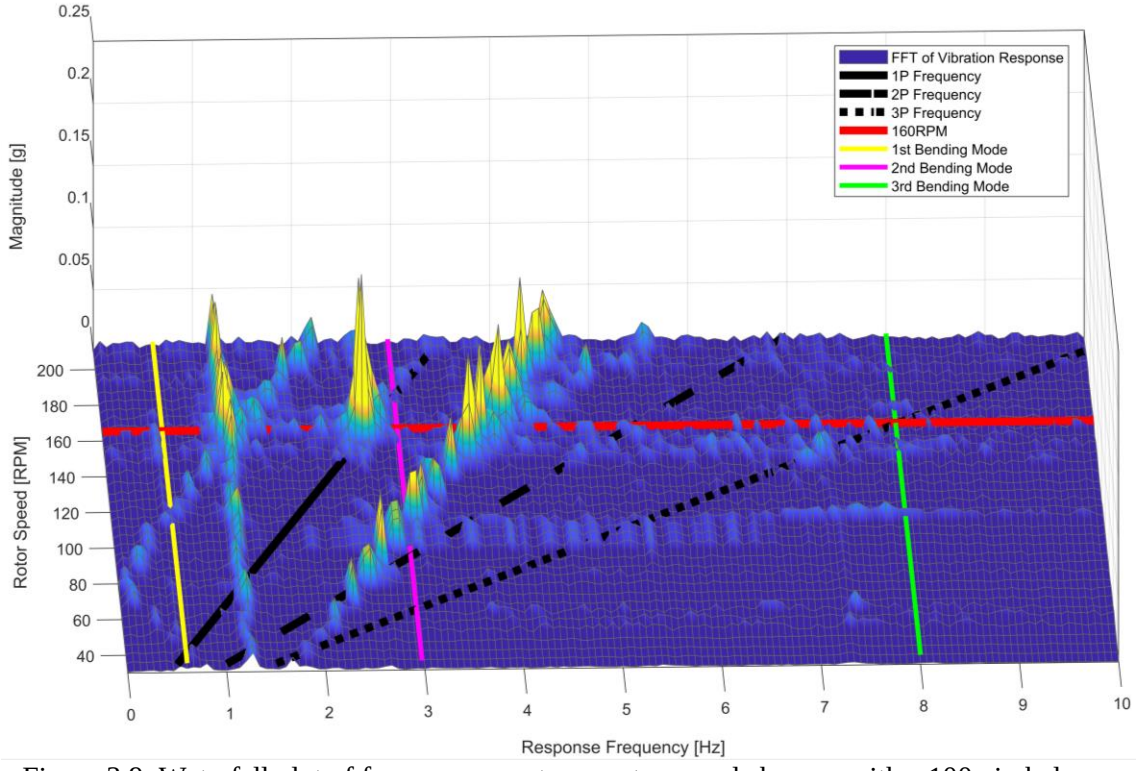


Figure 3.9: Waterfall plot of frequency spectra as rotor speed changes with a 100g imbalance

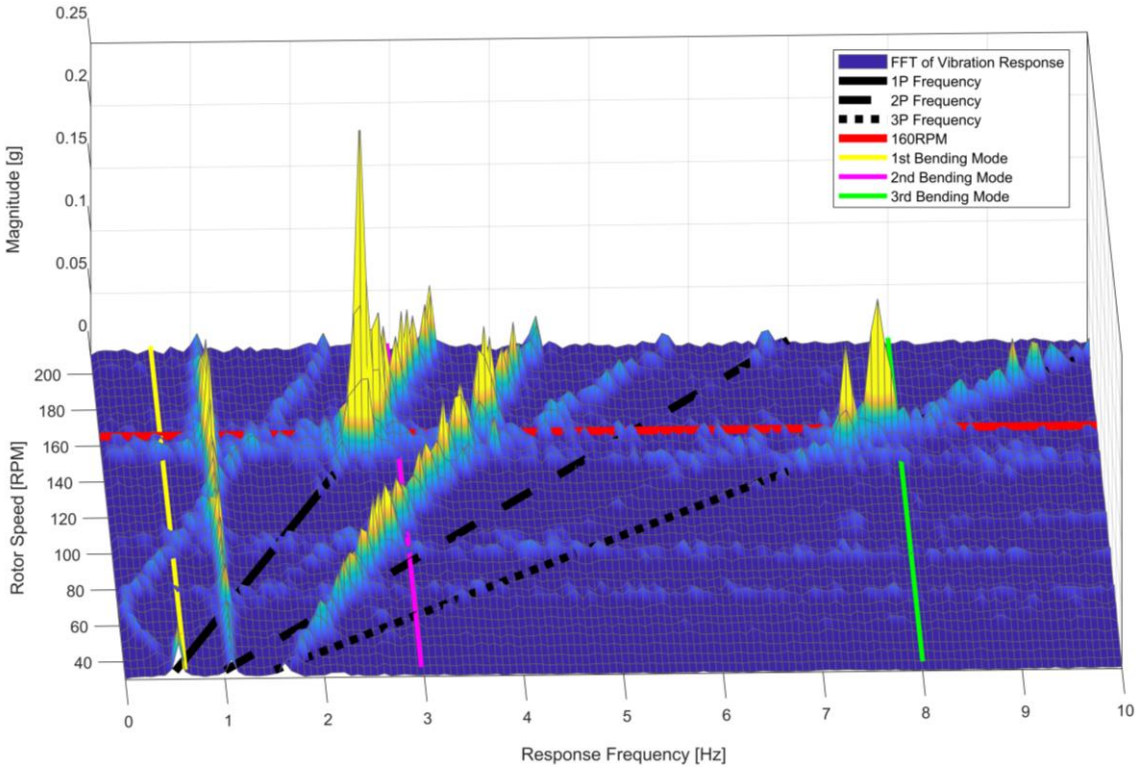


Figure 3.10: Waterfall plot of frequency spectra as rotor speed changes with a 200g imbalance

Analyses of the waterfall plots yield several interesting results. First, several vibrational peaks appear in all waterfall plots which do not correspond to any expected vibrational theory. One appears as a constant 1.1 Hz peak, while several others change with rotor frequency, appearing at (1.1 Hz + 1P), (-1.1 Hz + 1P), and (1.1 Hz – 1P). Several effects might cause this, ranging from dynamic effects such as tower nonlinearities to measurement issues including sensor aliasing. Regardless, the knowledge that these peaks are present allows for ignoring them in frequency analyses, thus minimizing their impact on a fault detection algorithm.

As shown in Figure 3.9 and Figure 3.10, the addition of a mass imbalance causes a clear peak at the 1P frequency once the operating speed reaches 140 RPM. This effect occurs as the forcing frequency aligns with the resonant frequencies of the tower bending modes, consistent with the theory presented in section 1.3. With the larger 200g imbalance, the shaking force becomes large enough to show a clear 1P frequency peak with the rotor speed outside the resonance region. The 200g imbalance also caused the blade to bend out of plane from the other two blades. This bending

motion resulted in a significant excitation at the 3P frequency in addition to the 1P excitations seen in Figure 3.9. This effect disagrees with the first theory of aerodynamic imbalances presented in section 1.2. It is thus theorized that the bending motion generated turbulent eddies within the rotor plane, giving rise to the rotational sampling of turbulence. This would cause a significant 3P component of tower vibrations, as shown in Figure 3.10. Further testing to investigate this effect is suggested; however, for the present work, it will not be investigated further. The bending modes of the tower's weak axis also clearly appear in these waterfall plots, as the 1P and 3P vibrations excite the 2nd and 3rd bending modes of the tower. The location of the peak on the waterfall plot may be used to estimate the bending mode frequencies: Table 3.3 lists the estimated mode frequencies and compares them to the previously created models.

Table 3.3: Tower weak axis bending modes as described by resonance over rotor ramp tests

Bending Mode [#]	Testing [Hz]	Katsanis Model [Hz]	Percent Diff. [%]	Gwon Model [Hz]	Percent Diff. [%]
1st	0.51	0.59	14.6	0.58	12.8
2nd	2.61	2.97	12.9	2.81	7.4
3rd	7.84	8.05	2.6	8.13	3.63

As shown, testing shows slight differences from the theoretical modes. This is expected, as both Katsanis' and Gwon's model assume a linear tower model; some error is therefore expected in real turbine operation. Furthermore, bending modes are estimated based on the relevant peak in tower vibrations; measurement error may thus also cause error in bending mode estimation.

To conclude, the best time-series-based measures for differentiating between a healthy rotor and an imbalanced rotor – using only accelerometer data – are RMS and Line Length values. A mass imbalance offsets these measures in a predictable direction towards a larger magnitude. For frequency-based analysis, the Fast Fourier Transform reliably shows both mass and aerodynamic imbalances with the rotor speed above 160 RPM. The fault detection algorithms developed for the present work will therefore apply time-series analysis using the RMS and Line Length values, and frequency-based analysis based on the FFT.

Chapter 4

CONDITION MONITORING AND FAULT DETECTION ALGORITHMS

In general, fault detection algorithms will contain two major components: a mathematical model of the turbine, and a decision-making algorithm analyzing the results of this model. A simple version of this is shown in Figure 4.1.

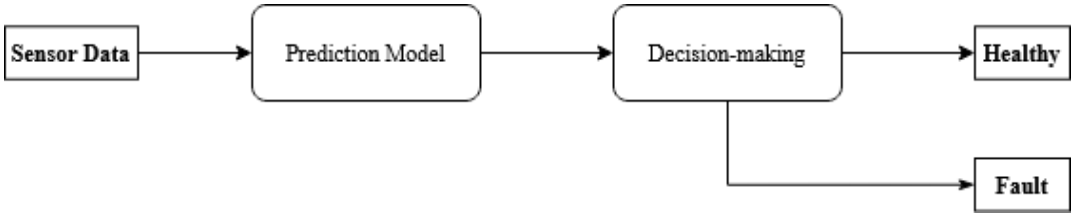


Figure 4.1: Flowchart describing the process flow of a typical fault-detection algorithm

Fault detection algorithms take place a step removed from traditional control systems. That is, the algorithm only affects the control system when a serious fault is detected. In a typical algorithm, parameters of the system are continuously monitored, and significant deviations from normal operation raise an alarm. The alarm either alerts a technician or automatically stops the system, depending on algorithm implementation and seriousness of the detected fault.

Application of a fault detection algorithm to a wind turbine must specifically consider the unique challenges present to the system – especially the fact that wind is an inherently stochastic phenomenon. Many models have been used to describe the random variations in wind speed, both in space and in time. The best-known model is the Von Karman wind turbulence model, which is the preferred model of the U.S. Department of Defense [23].

For stochastic systems, a common prediction model is the computation of a residual [24]. The residual is the difference between the expected and actual state of a given system, as illustrated by Figure 4.2. The expected state is created through any type of model; the most common form is a linear state-space model. For strongly nonlinear systems, such as those in rotor dynamics, a

linearized model must be created at the operating point. Variable-speed systems like the CPWT thus require the creation of numerous linearized models at each operating point. To avoid this, the residual-based prediction model used in the present work uses a nonlinear state estimating technique described in section 3.

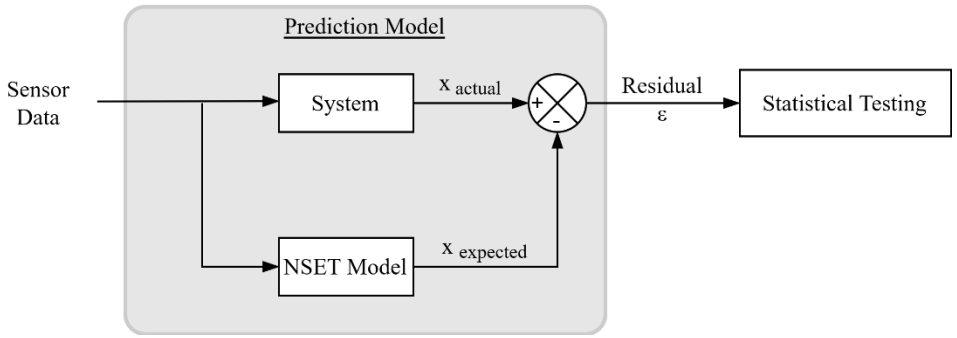


Figure 4.2: Block diagram for residual-based testing

This thesis applies three algorithms to the CPWT testing data collected in chapter 3. Both time-series and frequency-based methods are used. In all cases, algorithm application begins with a training phase, in which the algorithm "learns" the healthy operating state of the rotor. Then, new data is tested sequentially. The developed algorithms are tuned, and their performance is described, via the cross-validation study in chapter 5.

4.1 NSET+SPRT

The NSET+SPRT is a time-series algorithm. It uses the Nonlinear State Estimation Technique (NSET) as the prediction model to memorize and predict the state of the rotor, and the Sequential Probability Ratio Test (SPRT) for decision-making to decide on the presence of a fault. NSET is a condition monitoring algorithm used to detect deviations from the normal operating state of a system. It is part of a more general class of algorithms known as Similarity-Based Modeling (SBM). After a period of training, the algorithm can be trained to estimate the state of a process given new sensor measurements. A major advantage of this algorithm is its ability to detect both process faults and sensor faults (and differentiate between them). Without a secondary decision-making

algorithm, however, NSET would hold no ability to detect faults. This is solved with the SPRT. The SPRT is a powerful statistical testing tool that has been used for many applications, including quality control processes and statistical fault detection. This chapter outlines the theoretical backbone for both techniques and establishes methods for training the algorithm and using it for fault detection and condition monitoring.

4.1.1 NSET Background

Argonne National Laboratory and Florida Power Corporation developed their Multivariate State Estimation Technique (MSET) in the 1990s to model the Crystal River-3 nuclear power plant, located in Crystal River, Florida [25]. MSET modeled feedwater flow meters and successfully predicted degradation in measurements due to sensor surface fouling. A major benefit of adopting this system was its ability to detect incipient faults; that is, a slowly degrading sensor or component within a system. Since then, MSET has been used in many US Nuclear Reactors, and various other industries.

In 2012, Peng Guo and David Infield adapted this algorithm for monitoring wind turbine performance (which they called the Nonlinear State Estimation Technique, or NSET) [26], [27]. They applied the technique for monitoring the generator (via temperature measurements) and the nacelle (via tower vibration measurements). They created faults in the generator, which NSET was able to detect. They also showed that NSET was able to predict nacelle vibrations to a high degree of accuracy, but did not use it to detect faults.

4.1.2 NSET Theory

The Nonlinear State Estimation technique is a memory-based condition monitoring algorithm based on the formation of system state vectors, wherein each vector element is an expected sensor measurement. The technique is broken up into three major steps. First, a memory matrix is learned using historical sensor data. After the learning phase, the system may proceed into the detection phase. When new data is collected, a weighting vector is calculated. The estimated system state is

then the linear combination of all vectors multiplied by their weighted value. In a healthy system, the difference between the measured state and the estimated state will be small. However, should new data significantly deviate from historical observations (such as in the case of a process fault or sensor failure), NSET will produce an estimated state that is different from the measured state. This difference is quantified by computing the difference between the measured and estimated system state.

The data used to form system states must follow several criteria:

1. All sensors must be sampled at, or averaged to, the same data collection rate
2. For stochastic processes, the data collection rate should be averaged to a sufficiently large period such that random variations are filtered out
3. Measured parameters must have some level of interconnectedness

Following this, successive steps in time form a measurement vector composed of data from each sensor. For a system with *n* sensors, the following column vector is created:

$$\boldsymbol{X}(i) = [x_1(i), x_2(i), \cdots, x_n(i)]^T \tag{4.1.a}$$

For a set of historical operating data, a separate algorithm selects *m* measurement vectors to be included in a memory matrix. Then, the memory matrix $\boldsymbol{D}$ is the combination of all *m* measurement vectors:

$$\boldsymbol{D} = [\boldsymbol{X}(1), \boldsymbol{X}(2), \cdots, \boldsymbol{X}(m)] = \begin{bmatrix} x_1(1) & x_1(2) & \cdots & x_1(m) \\ x_2(1) & x_2(2) & \cdots & x_2(m) \\ \vdots & \vdots & \ddots & \vdots \\ x_n(1) & x_n(2) & \cdots & x_n(m) \end{bmatrix} \tag{4.1.b}$$

After forming this matrix, an estimation of the system state can be found by multiplying $\boldsymbol{D}$ by an *m*-dimension weighting vector $\boldsymbol{W}$.

$$\boldsymbol{X}_{est} = \boldsymbol{D}\,\boldsymbol{W} \tag{4.1.c}$$

As new measurement vectors ($\boldsymbol{X_{obs}}$) are taken, the residual $\boldsymbol{\varepsilon} = \boldsymbol{X}_{est} - \boldsymbol{X}_{obs}$ is computed. The weighting vector $\boldsymbol{W}$ is produced by minimizing the residual, produced via the equation:

$$\boldsymbol{W} = \left(\boldsymbol{D}^{\mathrm{T}}\,\boldsymbol{D}\right)^{-1}\left(\boldsymbol{D}^{\mathrm{T}}\,\boldsymbol{X}_{obs}\right) \tag{4.1.d}$$

This derivation may be found in ref. [28]. As discussed there, a major issue with this equation arose: as the memory matrix grew, the quantity $\boldsymbol{D^T D}$ quickly became ill-conditioned, preventing a proper inverse from being taken. Also, this quantity was not able to account for random fluctuations in sensor data. Thus, further development of NSET replaced the matrix multiplication with a nonlinear operator, signified by the $\otimes$ symbol. In ref. [25] and [28], the nonlinear operator was selected via a secondary algorithm; however, the operators used were not disclosed due to proprietary issues. For wind turbine analysis, ref. [26] and [27] used the Euclidean distance between the two vectors as the nonlinear operator:

$$\boldsymbol{X} \otimes \boldsymbol{Y} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{4.1.e}$$

When used, equation (4.1.d) becomes:

$$\boldsymbol{W} = \left(\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D}\right)^{-1}\left(\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{X}_{obs}\right) \tag{4.1.f}$$

Thus, the system state may be estimated by the full equation:

$$\boldsymbol{X}_{est} = \boldsymbol{D}\left(\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D}\right)^{-1}\left(\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{X}_{obs}\right) \tag{4.1.g}$$

Furthermore, for multiplying one or more matrices, the nonlinear operator functions as a linear matrix multiplication operation:

$$\begin{aligned}\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D} &= \begin{bmatrix}\boldsymbol{X}(1)\\ \boldsymbol{X}(2)\\ \vdots\\ \boldsymbol{X}(m)\end{bmatrix} \otimes \begin{bmatrix}\boldsymbol{X}(1) & \boldsymbol{X}(2) & \cdots & \boldsymbol{X}(m)\end{bmatrix} = \\ &= \begin{bmatrix}\boldsymbol{X}(1)\otimes\boldsymbol{X}(1) & \boldsymbol{X}(1)\otimes\boldsymbol{X}(2) & \cdots & \boldsymbol{X}(1)\otimes\boldsymbol{X}(m)\\ \boldsymbol{X}(2)\otimes\boldsymbol{X}(1) & \boldsymbol{X}(2)\otimes\boldsymbol{X}(2) & \cdots & \boldsymbol{X}(2)\otimes\boldsymbol{X}(m)\\ \vdots & \vdots & \ddots & \vdots\\ \boldsymbol{X}(m)\otimes\boldsymbol{X}(1) & \boldsymbol{X}(m)\otimes\boldsymbol{X}(2) & \cdots & \boldsymbol{X}(m)\otimes\boldsymbol{X}(m)\end{bmatrix}\end{aligned} \tag{4.1.h}$$

The major difference, then, is that the Euclidean distance is calculated rather than directly multiplying the vectors. This operator has been successful in monitoring several wind turbine subsystems, including the generator and the nacelle. As such, this first version of the NSET application uses it – however, additional exploration of other operators may be valuable. As discussed in ref. [29], the nonlinear operator may also be described as a similarity operation. In this, the level of similarity between a new observed input vector $\boldsymbol{X}_{\text{obs}}$ and the memory vectors is computed into a "similarity score vector," and transformed into a set of weighting factors corresponding to each vector present in $\boldsymbol{D}$.

4.1.3 Memory Matrix Formation

Formation of the memory matrix $\boldsymbol{D}$ is a critical prerequisite to live integration of NSET in a condition monitoring algorithm. As such, NSET includes a subset of algorithms dedicated to forming $\boldsymbol{D}$. The goal of these algorithms is to form $\boldsymbol{D}$ such that it encompasses the breadth of operational data for the system studied.

As discussed earlier, $\boldsymbol{D}$ must be formed such that $\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D}$ is well-conditioned. If this criterion is not fulfilled, then the inverse of $\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D}$, as described by equation (4.1.f), will not be properly computed. In this case, the estimated system state computed in (4.1.g) will be identical to the observed state vector, eliminating the NSET model's predictive power. Therefore, before developing any algorithm to form $\boldsymbol{D}$, a mathematical description of a well-conditioned matrix must be defined. A well-conditioned matrix may be numerically described by the matrix condition number. For an m-by-m matrix $\boldsymbol{C}$, the condition number may be approximated by computing the reciprocal of the 1-norm condition number, $\tilde{\Lambda}^{-1}$:

$$\tilde{\Lambda}^{-1} = \left[\max\left(C_{ij} n_i\right) * \max\left({C^{-1}}_{ij} n_i\right)\right]^{-1} \tag{4.1.i}$$

Equation (4.1.i) is written in summation notation with **n** as a 1-by-m vector of ones to simplify notation. Calculation of $\tilde{\Lambda}^{-1}$ is much more computationally efficient than finding the exact matrix

condition number, which requires singular-value decomposition of the matrix. Approximating the condition number via equation (4.1.i) is significantly faster than an exact calculation; for a 5-by-50 matrix, this approximation takes 25% of the time that the exact version requires. See Attachment A for details of this derivation and additional discussion. A well-conditioned matrix has $\tilde{\Lambda}^{-1}$ close to one; in practice, though, $\tilde{\Lambda}^{-1}$ quickly drops as the size of $\boldsymbol{D}$ increases. MATLAB begins to warn of an ill-conditioned matrix for $\tilde{\Lambda}^{-1} \leq 10^{-16}$. Therefore, for formation of a well-conditioned memory matrix, the following algorithms set a conservative threshold of $\tilde{\Lambda}^{-1} \geq 10^{-8}$. This is an important requirement; although $\left(\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D}\right)^{-1}$ may be computed when ill-conditioned, the algorithm will lack the ability to properly estimate the system's state. An added benefit of this requirement is that the NSET model estimates the system state significantly faster with a smaller memory matrix. With $\tilde{\Lambda}^{-1}$ = 1.02E-08 (corresponding to a 9-by-196 matrix), estimating two ramp tests from chapter 3 takes 16 seconds in MATLAB. On the other hand, with $\tilde{\Lambda}^{-1}$ = 4.89E-11 (corresponding to a 9-by-904 matrix), the estimation process takes 351 seconds and raises eleven false positives when conducting the statistical testing described in section 4.1.4 (compared to the zero false alarms raised during the first test). Thus, ensuring a well-conditioned memory matrix has benefits in saving processing time and increasing algorithm robustness.

After gathering training data, it must be preprocessed into a useful format. As stated earlier, data must be sampled or averaged to the same data collection rate. SCADA systems typically make decisions based on 5 or 10-minute averages [27]; however, due to the small quantity of data able to be collected, ten-second averages were taken instead. Then data is normalized so that the maximum value is 1, with normalization factors given by the final column of Table 4.1. This normalization tends to increase $\tilde{\Lambda}^{-1}$, allowing for the inclusion of more vectors in the memory matrix. Data is then organized into measurement vectors according to the first column of Table 4.1.

Table 4.1: Vector organization by sensor and associated normalization factor

Vector Position n	Sensor	Raw SR [Hz]	Description	Units	Normalization Factor
1	Anemometer	1	Wind Speed	[MPH]	24
2	Gear Tooth Hall Effect	1	Rotor Speed	[RPM]	300
3	Generator Voltage/Current	1	Power Output	[W]	1500
4	LifeLine (Accelerometer)	50	RMS, X-Dir	[16384*g]	4096
5			RMS, Y-Dir	[16384*g]	4096
6			RMS, Z-Dir	[16384*g]	18432
7			Line Length, X-Dir	[16384*g]	50,000
8			Line Length, Y-Dir	[16384*g]	50,000
9			Line Length, Z-Dir	[16384*g]	50,000

Data from sensors 1, 2, and 3 are processed into ten-second averages. Vibrational data is processed into two major forms: the RMS value and the Line Length signal. See chapter 3.2 for more details on these calculations.

Next, two algorithms select the most significant data for inclusion in the memory matrix. Figure 4.3 depicts the general process for forming the memory matrix.

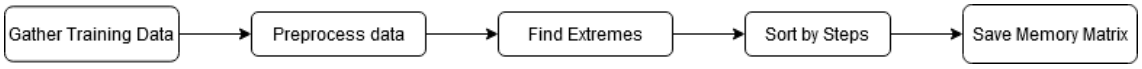


Figure 4.3: Overview of Memory Matrix Computation

The first algorithm selects vectors containing extreme measurement values. Vectors containing the minimum and maximum values from the wind speed and rotor speed sensors are selected, as described by ref. [25]. The second algorithm, a modified version of that found in ref. [26], divides the range of values from each sensor into $1/k_n$ steps (starting from k = 0.01). This results in 100 steps per sensor for the first iteration. The vector closest to each step increment is saved into a sensor-specific matrix $\boldsymbol{D}_n$. Once a vector is selected, it is removed from the available training data. After sorting, the matrices are combined such that $\boldsymbol{D} = [\boldsymbol{D}_1, \boldsymbol{D}_2, \dots, \boldsymbol{D}_n]$, and $\tilde{\Lambda}^{-1}(\boldsymbol{D})$ is calculated. If it is not above the threshold, the sensor n whose formed memory matrix has the

smallest $\tilde{\Lambda}^{-1}(\boldsymbol{D}_n)$ value has its number of bins reduced by adding $k_n = k_n + 0.01$, thus halving the number of steps per sensor each iteration. Figure 3 shows a graphical depiction of this algorithm. In practice, the $\tilde{\Lambda}^{-1}(\boldsymbol{D})$ exceeds the threshold when the matrix reaches around 200 vectors in length.

After a suitable memory matrix has been formed, the algorithm exits and saves $\boldsymbol{D}$ to system memory. Once this has been formed, the NSET model is ready to estimate the system state in a live implementation. However, the NSET model cannot recognize faults on its own. A decision-making algorithm must be imposed upon the result of this model. This thesis applies the Sequential Probability Ratio Test, or SPRT, to make these decisions – in line with the development in ref. [25] and [28].

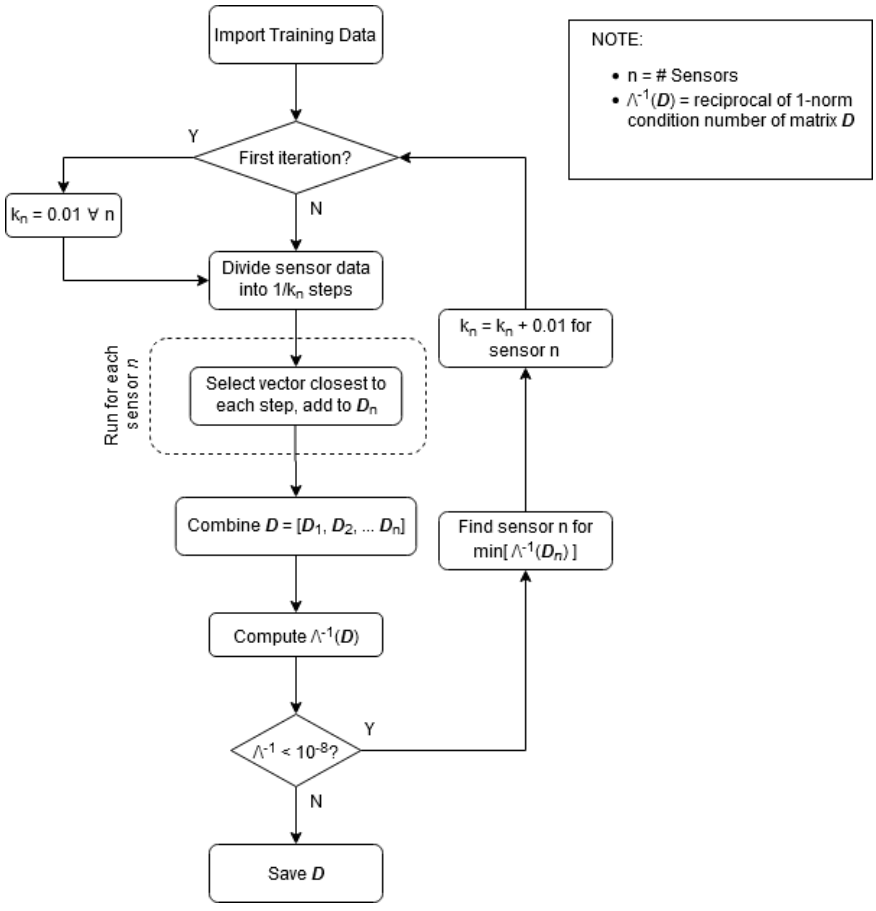


Figure 4.4: Visual flowchart describing the sorting algorithm for forming $\boldsymbol{D}$

4.1.4 SPRT Theory

The Sequential Probability Ratio Test, or SPRT, was developed by Abraham Wald in 1945 [30]. Although originally used for control of manufacturing processes, it has been since been to other cases, including testing human examinees and fault detection algorithms. The SPRT operates on the CPWT NSET model and allows for determining deviation from normal operation using statistical methods. As a sequential test, the SPRT does not operate on a fixed sample size; rather, it progressively adds additional data points to the test until a decision is reached. For each datapoint x_i and possible fault j, the SPRT chooses between one of three possibilities:

1. Accept the null hypothesis H_0 (No fault detected)
2. Reject the null hypothesis H_0, and assume H_j to be true (Fault j detected)
3. Collect another datapoint $i+1$

The SPRT operates on the residual ε_i between the observed and expected accelerations:

$$\varepsilon_i = x_{est,i} - x_{obs,i} \tag{4.1.j}$$

where the acceleration x is the set of Y-axis Line Length accelerations as described in chapter 3.2, and the estimated state is generated using the NSET algorithm. SPRT hypotheses are formulated similar to ref. [31], which developed a health monitoring system for radio-frequency-based wireless sensor systems. The SPRT tests the probability that the datapoint ε_i lies in the distribution specified by each hypothesis. Each of these hypotheses is outlined in Table 4.2. All hypotheses assume that ε is from a normal distribution – an assumption discussed in Appendix B. These assumptions proved true in some cases, but false in others; despite that, the algorithm is ultimately successful in operation. Thus, the final algorithm will proceed with this assumption.

Table 4.2: List of hypotheses used in the SPRT

Hypothesis	Mathematical Description	Physical Description	Causes
H_0	$\bar{\varepsilon}_{test} = 0$ $\sigma(\varepsilon_{test}) = \sigma(\varepsilon_{mem})$	System is operating normally	–
H_1	$\bar{\varepsilon}_{test} = +M$ $\sigma(\varepsilon_{test}) = \sigma(\varepsilon_{mem})$	Average vibrations are smaller than normal	Rotor is parked Large drops in wind speed
H_2	$\bar{\varepsilon}_{test} = -M$ $\sigma(\varepsilon_{test}) = \sigma(\varepsilon_{mem})$	Average vibrations are larger than normal	Rotor fault is present Higher winds than memorized
H_3	$\bar{\varepsilon}_{test} = 0$ $\sigma(\varepsilon_{test}) = V\sigma(\varepsilon_{mem})$	Average vibrations fluctuate more than normal	Sensor Error
H_4	$\bar{\varepsilon}_{test} = 0$ $\sigma(\varepsilon_{test}) = \frac{1}{V}\sigma(\varepsilon_{mem})$	Average vibrations fluctuate less than normal	Sensor Error

M and V are parameters that allow the algorithm's sensitivity to faults to be tuned. M is the system disturbance parameter and is defined by $M = m * \sigma(\varepsilon_{mem})$, where *m* is set by the operator. V is the variation factor. As hypotheses H_3 and H_4 do not indicate that tower vibrations are larger or smaller than normal, they are not used in raising alarms for detecting rotor unbalance. However, as discussed in ref. [32], they are commonly indicative of wiring or sensor issues and are thus included in the algorithm.

The SPRT makes decisions based on the likelihood ratio between two possibilities: that the null hypothesis is true, or that the j^{th} alternative hypothesis is true. The probability that a residual ε_i is a part of the hypothesis H_j defined by mean μ and standard deviation σ is given by the normal distribution probability density function:

$$\Pr(\varepsilon_i \mid H_j) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{\varepsilon_i - \mu}{\sigma}\right)^2\right] \tag{4.1.k}$$

The likelihood ratio, or the ratio between $\Pr(\varepsilon_i \mid H_j)$ and $\Pr(\varepsilon_i \mid H_0)$, is then calculated for each data point. New data points are included by multiplying their ratios:

$$LR_j = \frac{Probability\ of\ sequence\ \{\varepsilon_n\}\ given\ H_j\ true}{Probability\ of\ sequence\ \{\varepsilon_n\}\ given\ H_0\ true} = \prod_{i=1}^{n} \frac{\Pr(\ \varepsilon_i \mid H_j\ )}{\Pr(\ \varepsilon_i \mid H_0\ )} \tag{4.1.l}$$

Finally, the SPRT index is calculated as the natural logarithm of the likelihood ratio:

$$\text{SPRT}_\text{j} = \ln(LR_j) \tag{4.1.m}$$

In simplifying (4.1.m), the computational power necessary is reduced. For the first two alternative hypotheses, the SPRT index is reduced to equation (4.1.n) and (4.1.o) [31].

$$\text{SPRT}_1 = \frac{M}{\sigma^2} \sum_{i=1}^{n} \left(\varepsilon_i - \frac{M}{2} \right) \tag{4.1.n}$$

$$\text{SPRT}_2 = \frac{M}{\sigma^2} \sum_{i=1}^{n} \left(-\varepsilon_i - \frac{M}{2} \right) \tag{4.1.o}$$

For each new datapoint added to ε_n, the SPRT index is compared against two thresholds. These thresholds are defined by two statistical parameters: the false alarm probability α and the missed alarm probability β.

$$A = \ln\left(\frac{\beta}{1-\alpha}\right), \qquad B = \ln\left(\frac{1-\beta}{\alpha}\right) \tag{4.1.p}$$

The decisions discussed earlier are made based on the value of the SPRT index relative to thresholds A and B. They are summarized in Table 4.3.

Table 4.3: Decisions made by the SPRT given the index and thresholds A & B.

Value of SPRT_j	**Decision**
$\text{SPRT}_\text{j} < \text{A}$	Accept H_0 (no fault detected)
$\text{SPRT}_\text{j} > \text{B}$	Reject H_0 and accept H_j (possible fault detected)
$\text{A} < \text{SPRT}_\text{j} < \text{B}$	No decision (collect more data)

When the SPRT decides on hypothesis j, the occurrence is logged, the $SPRT_j$ is reset to 0, and sequential testing continues. A visualization of the SPRT detecting a fault is shown in Figure 4.5.

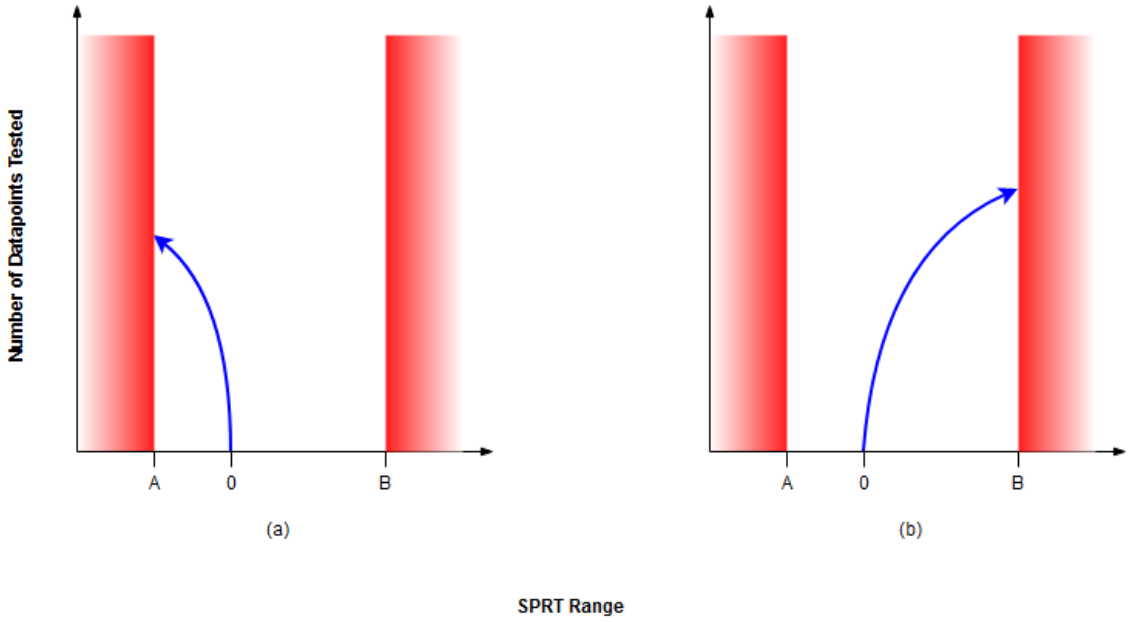


Figure 4.5: Example of SPRT concluding (a) the null hypothesis (b) the alternative hypothesis j

The SPRT described here is adjusted using four parameters- m, V, α, and β. These parameters are best adjusted through a cross-validation study. Such a study is completed in chapter 5.

4.1.5 Application of NSET+SPRT

The data used to train the algorithm must adhere to two characteristics:

1. The data must encompass the full operating range of the turbine
2. There must be enough data that the residual standard deviation must be equal for healthy training data and healthy testing data

For 1 to be true, the set of training data for all operating parameters must include the set of data to be tested. A 2-dimensional visualization of this is shown in Figure 4.6.

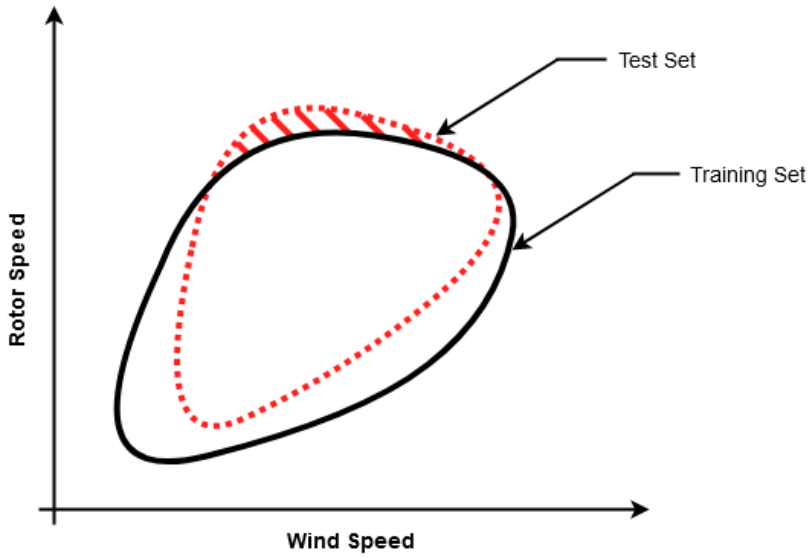


Figure 4.6: 2D Visual of training set and testing set of data. The shaded region is data outside the range memorized by the algorithm; it represents possible modeling errors

To meet this requirement, all datasets used to train and test NSET+SPRT were collected by completing a ramp test from 30-210RPM over 30 minutes. All tests were completed with 10-15 mph winds.

Testing shows that for 2 to be true, a significant amount of data must be memorized, which is a major drawback of the NSET. In general, studies applying these two algorithms have months of data to work with. However, this study is particularly limited because an operator must be present to run the turbine, and there is often only wind ideal for data-collection for a short period during the day. To determine the optimal amount of data necessary for training, a number of ramp tests

(as described by chapter 3) are used to train, and then test, the NSET model. Table 4.4 shows the number of datasets used to train the algorithm, along with the training and testing data's residual standard deviation.

Table 4.4: Statistical characteristics for various numbers of datasets used to train NSET+SPRT

Number Datasets Learned $n_{learned}$	Avg Std Dev of Residual for Training Data $\sigma(training)$	Avg Std Dev of Residual for Testing Data $\sigma(testing)$	Percent Difference %
1	3.759 E-03	5.388 E-02	1333.2
4	8.968 E-03	2.682 E-02	199.0
7	9.294 E-03	1.602 E-02	72.4
9	1.154 E-02	1.190 E-02	3.05

The percent difference as a function of $n_{learned}$ is also shown in Figure 4.7.

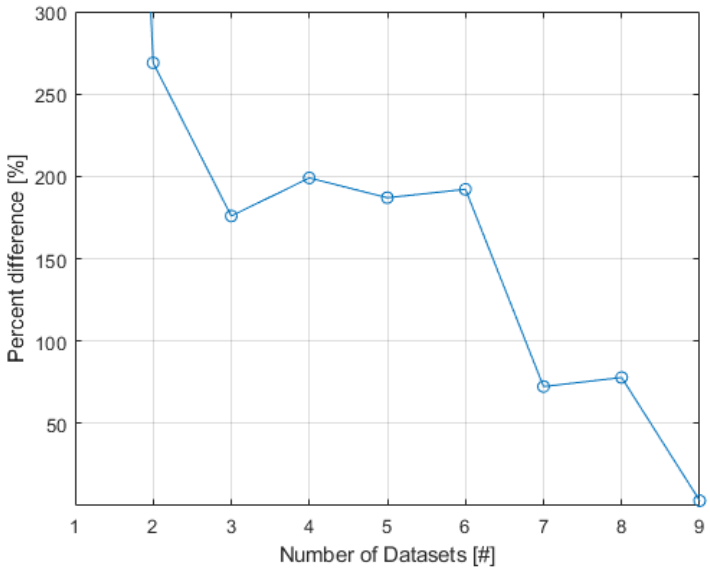


Figure 4.7: Percent difference versus number of datasets learned

As shown, including more datasets in the training set substantially reduces the percent difference. This occurs for two reasons: an increase in $\sigma(training)$ as the algorithm memorizes less of the entire set of data, and a reduction in $\sigma(testing)$ as the algorithm more accurately estimates the system state under other operating conditions. Memorizing nine sets of data presents the lowest percent difference (3.89%), which corresponding to 4.5 hours of data.

With nine sets of data memorized, the combined NSET+SPRT algorithm can differentiate between balanced and unbalanced operating data to a high degree of accuracy, allowing for robust estimation of the system operating state. An example of the test operating on balanced and unbalanced data may be seen in Figure 4.8. The Y-axis Line Length is the best parameter for monitoring, as it results in the highest rate of correct fault decisions and the lowest rate of incorrect fault decisions; therefore, it is the residual value used for SPRT testing.

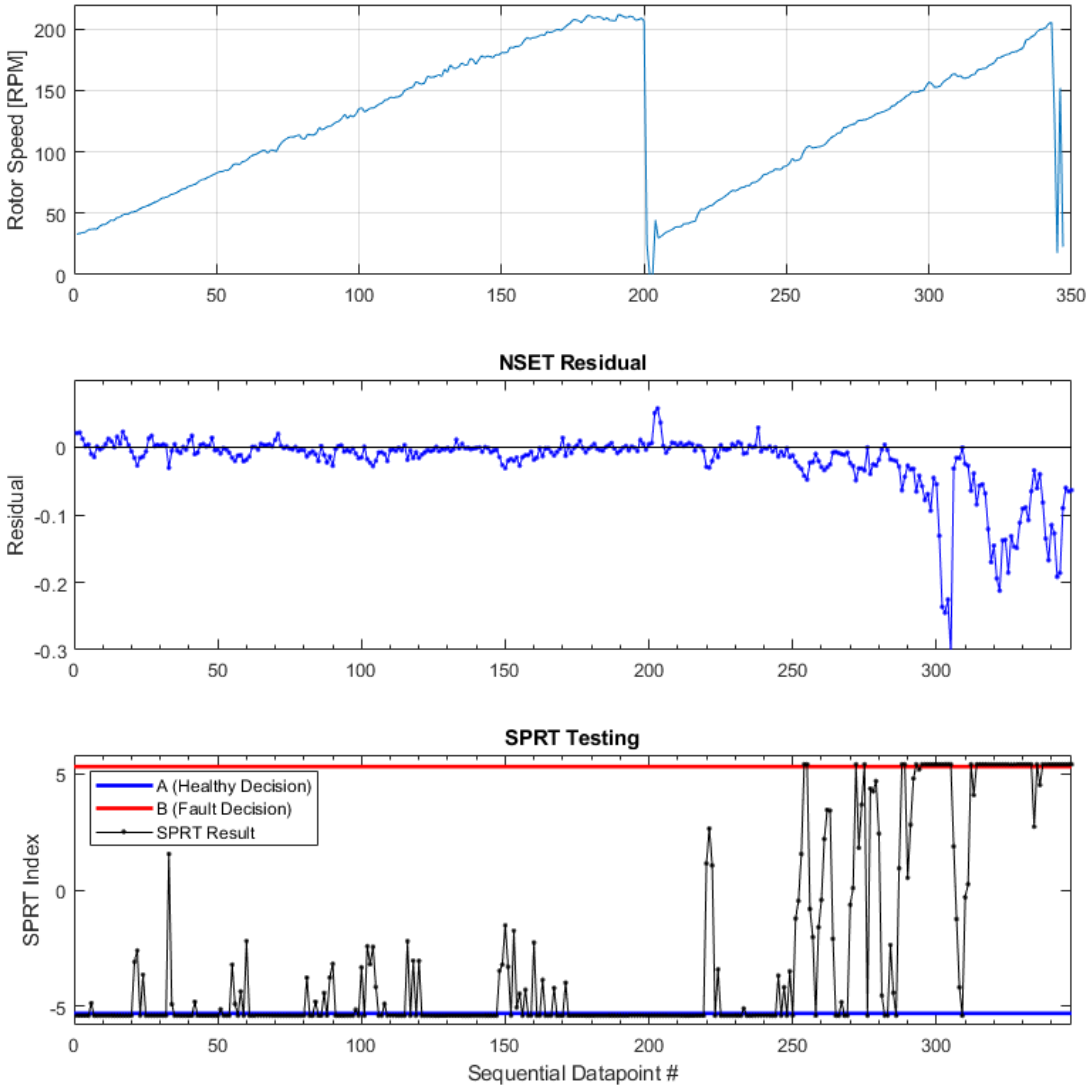


Figure 4.8: NSET+SPRT testing of balanced and unbalanced data. Unbalanced data begins at x = 204. The Y-axis Line Length parameter is used for monitoring.

4.2 FFT: Adaptive Threshold (AFFT)

The second algorithm designed for the present work uses the Fast Fourier Transform (FFT) to memorize the frequency response of the rotor. As described by the waterfall plots in chapter 3.3 above, the peaks in the frequency spectra of nacelle vibrations significantly shift as rotor speed changes. As the CPWT is a variable-speed machine, any frequency-based method must account for this change in frequency spectra. This algorithm solves this with an adaptive threshold [33] – that is, a frequency-based threshold is set for each predefined increment in rotor speed. First, the rotor speed bins are set; for this algorithm, bins are set in 5RPM increments from 30 to 210RPM. Then, the healthy data is organized into each bin, and divided into sets of 1024 data points. The FFT is computed for each set, and the maximum amplitude for each frequency within a set frequency width w is learned:

$$A_{ij,threshold} = K_{thr} \max\left([A_{j-w}, A_{j+w}] \right) \tag{4.2.a}$$

where A_j corresponds to the amplitude of the FFT output at frequency j, and $A_{ij,threshold}$ corresponds to the threshold in rotor speed bin i and at frequency j. K_{thr} is an algorithm tuning parameter that dictates the magnitude of vibrations greater than learned necessary for the threshold to be crossed. For data within a given rotor speed bin, this process forms an alarm threshold as shown in Figure 4.9.

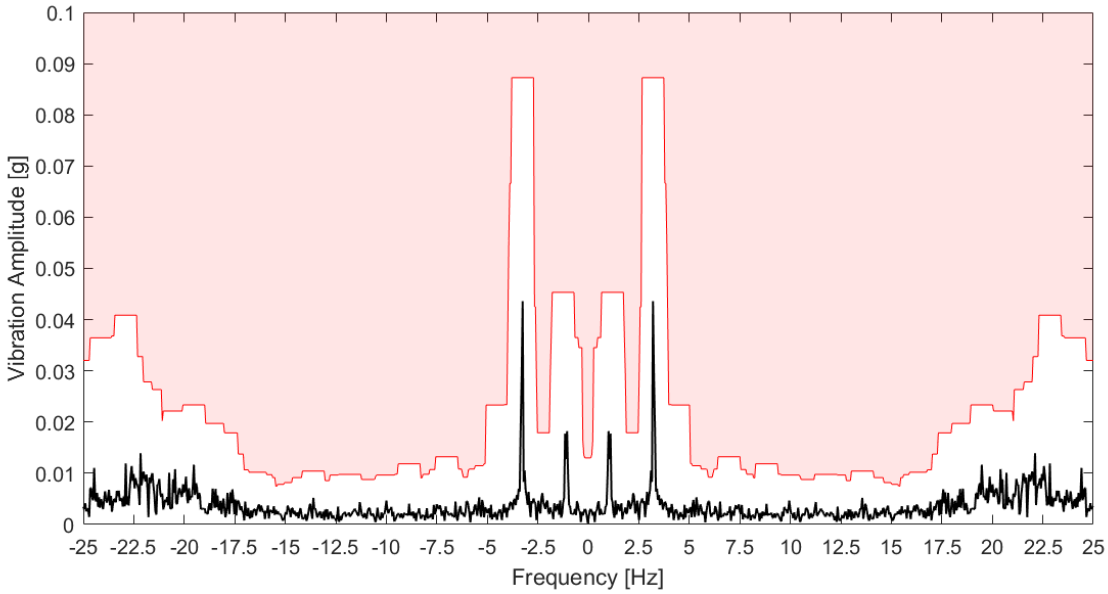


Figure 4.9: Adaptive threshold for data with average rotor speed $\Omega_{\text{avg}} = 130$ RPM.

Then, the frequency spectra for a set of data with rotor speed Ω_{avg} may be compared to the adaptive threshold according to the following logical inequality [33].

$$IF\ \left[\Omega_{min,i} < \Omega_{avg} < \Omega_{max,i}\right]\ THEN\ [\ A_j < A_{threshold,ij}] \tag{4.2.b}$$

If the above logic is *not* true, then a possible fault is logged. The frequency at which the FFT output surpasses the threshold is logged, and saved in terms of the multiple of rotor speed frequency:

$$f_{log} = \frac{60\ f_{alarm}}{\Omega_{\text{avg}}} \tag{4.2.c}$$

This allows for determining the potential source of a fault. If f_{log} = 1, then a mass imbalance is likely, while if f_{log} = 3, then an aerodynamic imbalance is likely. This algorithm is also prone to noise-related false positives; an example of one is shown in Figure 4.10.

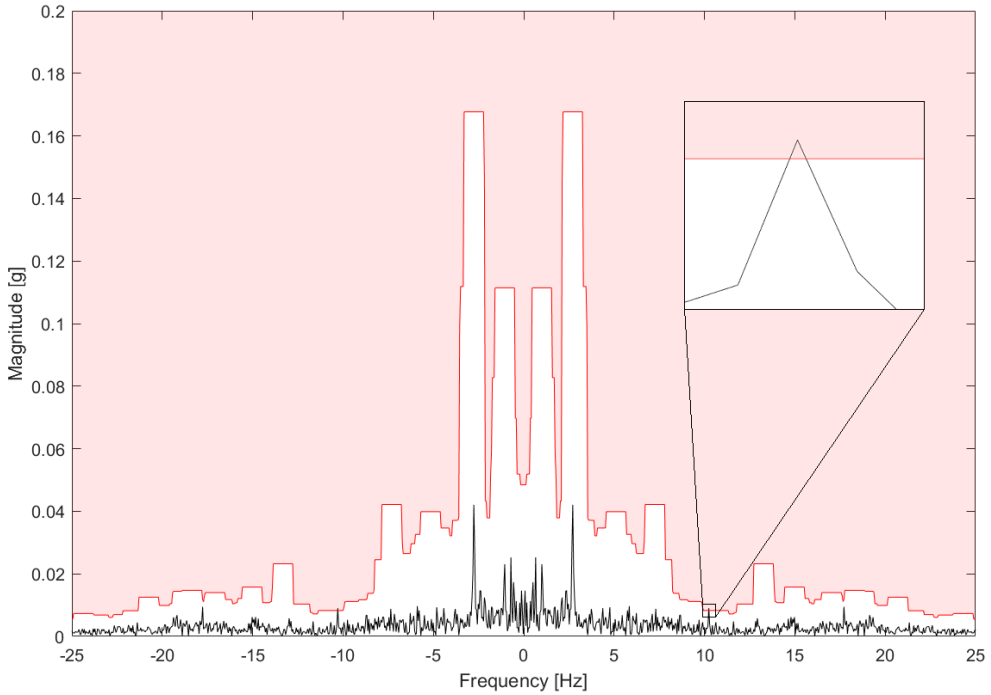


Figure 4.10: Example of an erroneous fault prediction at 10Hz. Note that an identical false positive is logged at -10 Hz.

These false positives may be avoided by only logging a fault when the adaptive threshold is exceeded within expected frequency ranges. For the present work, the desired faults to be detected are mass and aerodynamic imbalances resulting from blade damage. As discussed, these appear at the 1P and 3P frequency, respectively. Therefore, a fault is only logged when equation (4.2.b) is not true *and* when:

$$\left[\, 0.75 < \ f_{log} \ < 1.25 \,\right] OR \left[\, 2.75 < \ f_{log} \ < 3.25 \right] \tag{4.2.d}$$

With this secondary filter, applying the AFFT to data collected in chapter 3 results in zero false positives detected, while a 1P or 3P vibration frequency is quickly detected. This algorithm works well in the case of slow rotor speed changes. However, as discussed in section 1.2, if the rotor speed significantly changes throughout a single set of data used to take the FFT, then frequency peaks will shift along the frequency domain. This may cause the adaptive filter to log a fault when none is present; however, for all cases of functional operation, this shift was not significant enough to impact algorithm performance.

4.3 FFT: Order Analysis (OFFT)

The final algorithm introduced by this thesis also utilizes the Fast-Fourier Transform (FFT) and corrects the problems detailed in section 1.3. The Cal Poly Wind Turbine is a variable-speed turbine; as such, the rotor speed may change significantly by the time enough data is collected for a single FFT computation. This causes blurring of spectral lines in a single frequency spectrum analysis. A common approach to fix this is known as *order analysis*, wherein the vibration sample is resampled from constant time-steps to a constant number of samples per shaft revolution. Ref. [13] and [34] separately used order analysis to detect faults in wind turbines. Ref. [34] used order analysis to detect mass unbalance faults; their algorithm uses the complex vector resulting from the FFT as a condition indicator for detecting faults. Healthy data is used to define a circular threshold on the complex plane; new complex vectors whose endpoints lie within this threshold indicate healthy data, and vectors whose endpoints lie outside the threshold indicate a possible fault. Section 4.3.1 briefly discusses the mathematical basis for the algorithm and 4.3.2 describes the specific challenge of applying it to the Cal Poly Wind Turbine.

4.3.1 Order Analysis Theory

The algorithm requires that the acceleration signal be sampled at equal steps of rotor angle. As discussed in chapter 3, the data-collection system currently logs acceleration in constant timesteps. Therefore, the algorithm begins by interpolating the acceleration signal to the constant angle-step domain:

$$a(k_t * t) \rightarrow a(k_\theta * \theta_r) \tag{4.3.a}$$

After interpolation, the FFT of the resulting dataset is taken and the 1P component of vibration is extracted. Unlike the method discussed in Section 4.2, the FFT is left as a complex vector. As noise often causes a shift in frequency peaks resulting from the FFT, a range of frequencies close to the 1P frequency are extracted. A particular benefit of taking the FFT post-interpolation is the 1P frequency is trivial to identify and extract.

Healthy data is then used to train the algorithm. The algorithm uses a circular threshold: any new vector whose endpoint lies outside this circle is flagged as a fault. Figure 4.11 shows an example of this learned threshold.

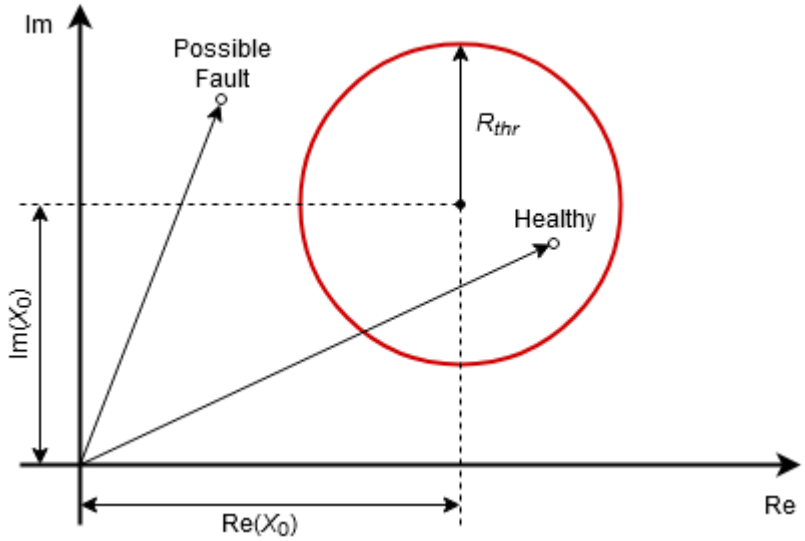


Figure 4.11: Plot of Order Analysis threshold, with a healthy vector and faulty vector shown.

To calculate the threshold, healthy 1P frequencies are assembled into the vector $\boldsymbol{X_{mem}}$. Then, the average real and imaginary value of each complex vector is taken and assembled into the center point $\boldsymbol{X_0}$:

$$Real(\boldsymbol{X}_0) = \frac{\sum_{i=1}^{N} Real(\boldsymbol{X}_{mem})}{N} \quad , \quad Imag(\boldsymbol{X}_0) = \frac{\sum_{i=1}^{N} Imag(\boldsymbol{X}_{mem})}{N} \qquad (4.3.b)$$

The radius of the threshold circle R_{thr} is also determined by calculating the standard deviation of the real and imaginary components of $\boldsymbol{X_{mem}}$. Then, R_{thr} is the maximum of these two values:

$$R_{thr} = K_{thr} * \max\{\sigma[Re(\boldsymbol{X}_{mem})], \sigma[Im(\boldsymbol{X}_{mem})]\} \qquad (4.3.c)$$

The factor K_{thr} is a sensitivity parameter that may be tuned by the operator; ref. [34] recommends values between 3.5 and 4.5. After defining the threshold circle, new data may be tested. For each new 1P frequency vector $\boldsymbol{X_i}$, the scalar $dX = |\boldsymbol{X_i} - \boldsymbol{X_0}|$ is computed. Then, a

possible fault flag is raised if $dX > R_{thr}$. The monitoring system throws an alarm if this flag occurs more than three times in a row.

4.3.2 Order Analysis Application

The mathematical backbone for this algorithm seems simple; however, several factors complicate application. First, the interpolation described by equation (4.3.a) is made challenging by the fact that rotor speed is taken once per second. As the acceleration is sampled at 50Hz, the rotor speed must be assumed constant for each of the 50 data points taken for each rotor speed recorded. Thus, this method's accuracy decreases during large rotor speed changes. A general process flow of this interpolation can be found in Figure 4.12.

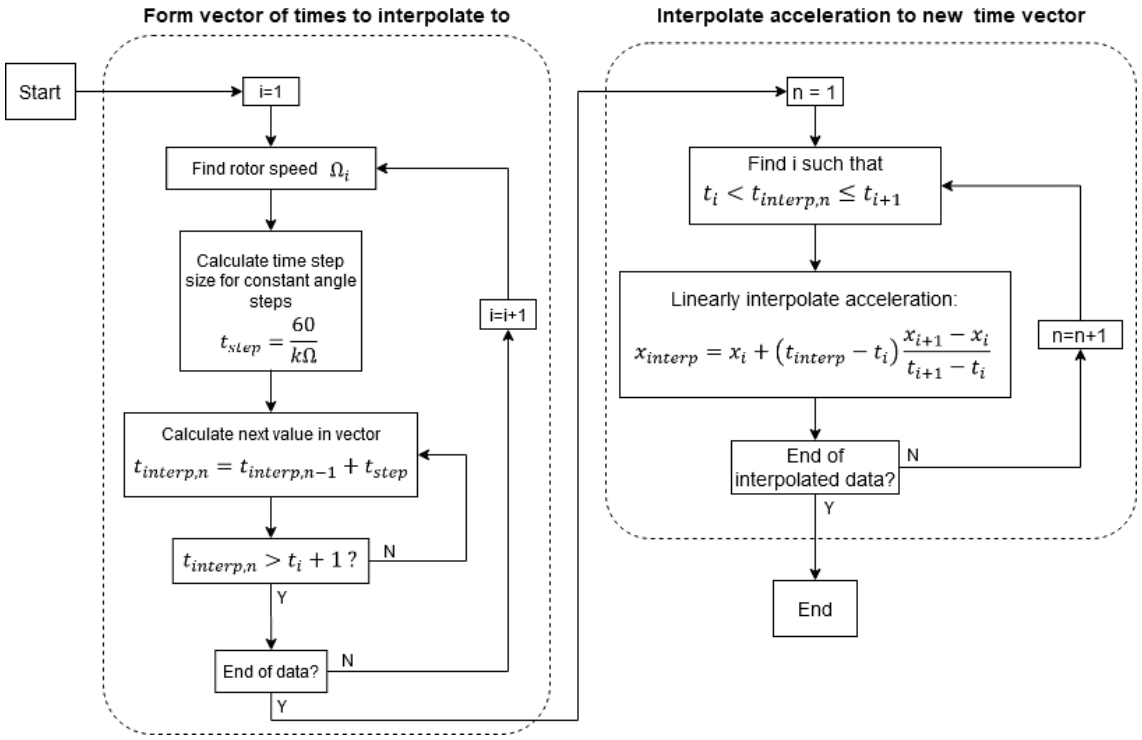


Figure 4.12: Interpolation from constant time-step domain to constant angle-step domain

This interpolation must be completed such that the resulting signal sufficiently captures local maxima and minima of the vibration signal. Trial and error shows that a sample rate of 16 data

points per shaft rotation is sufficient for this capture. Figure 4.13 shows a section of data resulting from this interpolation.

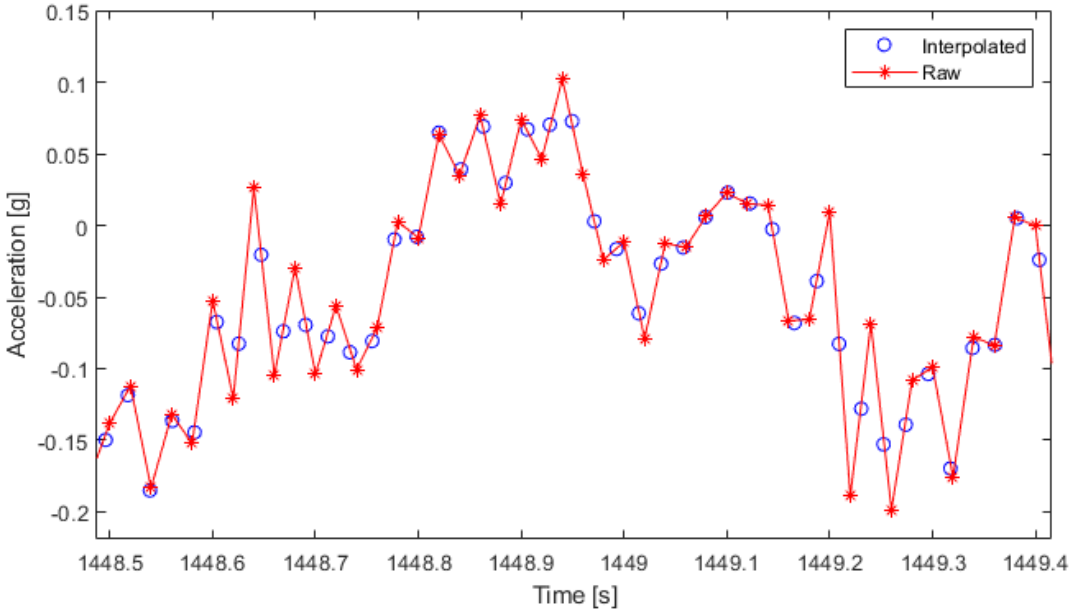


Figure 4.13: Raw acceleration and interpolated acceleration plotted in the time domain

After interpolation, the FFT is taken. Ref. [34] recommends at least 100 seconds of data be used for a single FFT. This corresponds to 8000 data points at an operating speed of 300 RPM, or the maximum speed of the rotor. The FFT is most efficient when using N data points such that N is a power of 2; therefore, analysis will use 8192 datapoints per FFT. If the 1P frequency component is outside the threshold greater than three times in a row, an alarm is declared. The final flowchart for the algorithm is described by Figure 4.14.

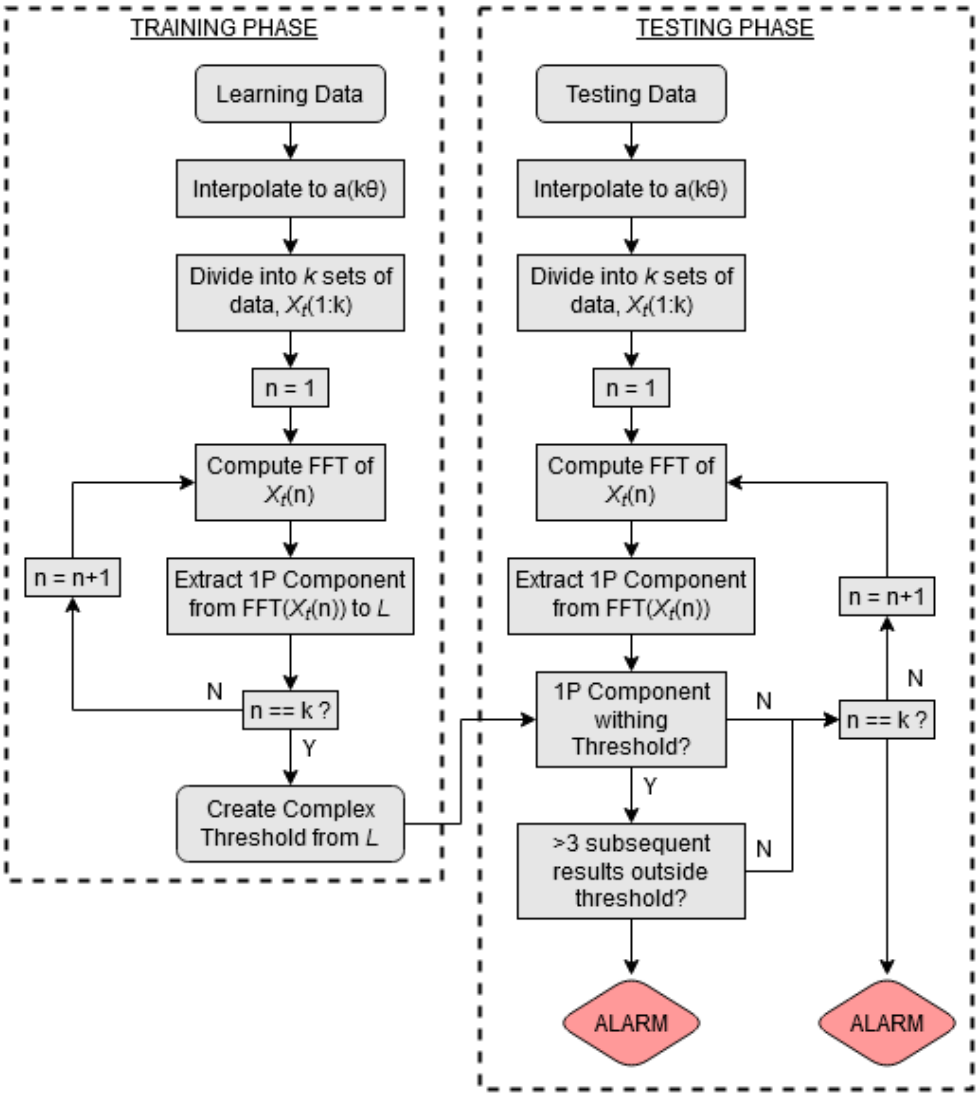


Figure 4.14: Training and testing flowchart for the Order Analysis algorithm

In practice, the need to interpolate the acceleration dataset before fault detection greatly increases the total computation time. In the future, the computation time may be significantly reduced by implementing the order analysis in hardware. This may be done by using the hall-effect sensors currently monitoring rotor speed. Then, the sensor could record accelerations each time the magnet passes the hall-effect sensor. This is the approach taken in ref. [13].

4.4 Other Methods

In pursuit of a robust fault detection algorithm, one other method was developed that was not used further, as it was not suitable for implementation within the scope of the current project. Although this algorithm was deemed unsuitable for the current study, it is potentially useful if the problems discussed herein are addressed.

4.4.1 LSCh

The Load Susceptibility Characteristic, or LSCh, was developed for turbine monitoring by ref. [37] and found by the present work in ref. [36]. For this method, data is averaged to predefined intervals, and a best-fit linear regression is performed between power output and RMS vibrations:

$$Power = a * RMS + b \tag{4.4.a}$$

In ref. [37], averages are taken every 10 minutes, and 1000 samples are used for each linear regression. As such, this algorithm requires a significant amount of data, which is the major limiting factor for implementation on the LifeLine. This method was used to diagnose generator bearing faults in ref. [37]; there, they had access to roughly 480 hours of operating data. In this case, each regression analysis used 17 hours of operating data. As a result, 28 sets of parameters (A, B) were collected. In contrast, data considered for the present work was just 20 hours in length. Thus, this algorithm is better suited for long-term monitoring of a wind turbine without operators present. This method does show some promise; however. Figure 4.15 shows the result of regression analysis for all balanced and unbalanced data, and Table 4.5 shows the exact parameters calculated.

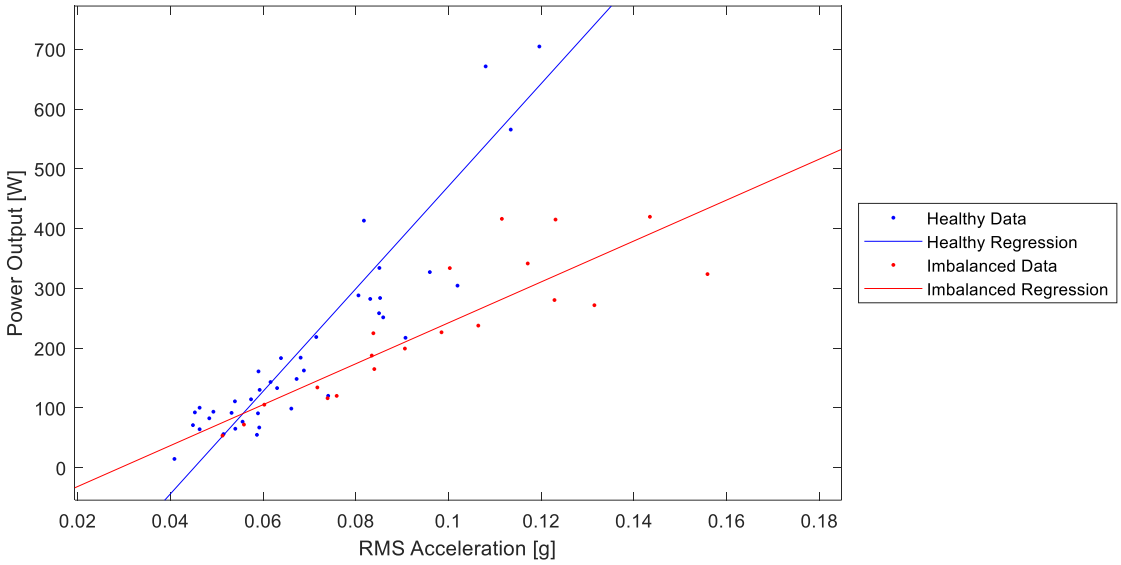


Figure 4.15: Regression results for healthy and imbalanced data

Table 4.5: Regression parameters for Figure 4.15

Component	Healthy Data	Faulty Data	Percent Diff.
Slope	8579.8	3423.1	85.9
Intercept	-387.12	-99.953	117.9

As a result of the significant amount of data necessary, this method will not be used further in the present work. However, if the CPWT is later allowed to run autonomously, this method should be considered for implementation.

Chapter 5

CROSS-VALIDATION STUDY

In many model validation and machine learning techniques, a cross-validation study is useful for both selecting between models and for tuning the technique used. For example, ref. [31] used a cross-validation study to tune the parameters of the SPRT used for detecting faults in wireless sensors. The present work, therefore, uses the study to tune each of the algorithms discussed in chapter 4. Furthermore, several parameters are defined that allow for comparing between each algorithm. Given the significant differences between each algorithm, these comparisons are not perfect – a limitation which is discussed at the end of the chapter.

5.1 Study Setup

A cross-validation study divides data into sets. Some of the sets are used to train the algorithms, and some are used to test the algorithm. A typical cross-validation study considers all data collected to be part of a single set, and arbitrarily divides this into training and testing sets. Special care must be taken when training each algorithm in the present work, however: to properly train them, the training data must represent the full operating range of the wind turbine. If this is not true, significant errors may develop when rotor speed and wind speed data are outside the range of training data. Section 4.1.5 further details this within the context of training the NSET+SPRT. Therefore, datasets are taken by completing a ramp test as described in chapter 2: a 30-minute ramp from 30 to 210 RPM, done by incrementing 1 RPM every 15 seconds. For this study, ten healthy runs and four faulty runs were collected. The four faulty runs were created by duct-taping a 200g sheet of aluminum to the turbine blade. All data is also trained with a wind speed between 10-15MPH and approximately westerly wind, to minimize the possibility of vibrational changes due to high winds or strong-weak axis interactions.

One powerful cross-validation method is known as a k-fold cross-validation method. In this, data is divided into k sets. A single set is used to train the algorithms, and the other k-1 sets are used to test each algorithm. As data is already divided into ten runs, the version used would thus be a 10-fold study. Although a valid method, this is problematic for the CPWT system as too small a training dataset leads to significant error. As discussed in section 4.1.5, model prediction becomes feasible with around six healthy sets trained. Another powerful method is known as Leave One Out (LOO). With the data already divided into sets, this method thus involves training each algorithm using nine sets of healthy data and testing the other healthy set. As shown by Figure 5.1, the 10-fold and LOO cross-validation studies are modified by testing the faulty dataset in all runs – the faulty dataset is never used to train the algorithm. This is necessary as, in practice, using faulty data to train the algorithms results in significant errors, and prevents accurately assessing algorithm performance.

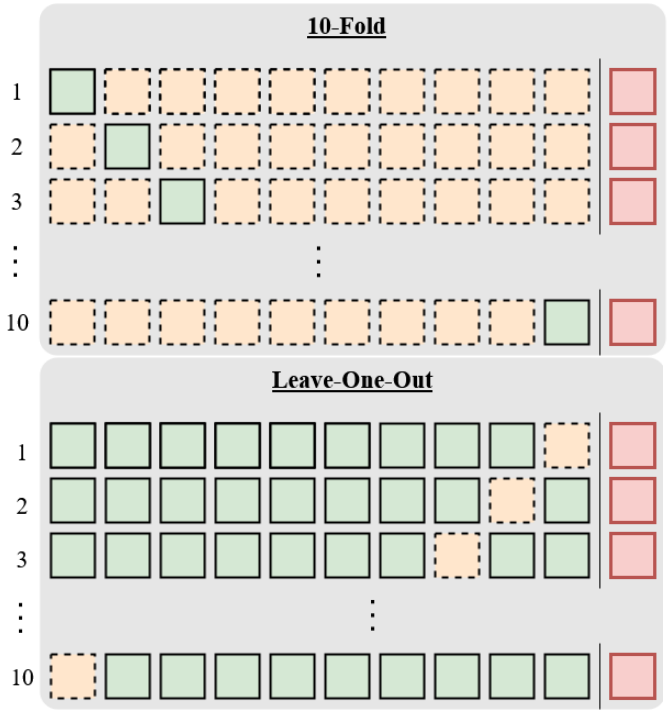


Figure 5.1: Visual overview of 10-fold and Leave-One-Out cross-validation studies. The green solid squares represent training datasets, the orange dashed squares represent testing datasets, and the red squares represent faulty datasets.

5.2 Parameter Selection

As discussed in chapter 4, each algorithm uses unique parameters that determine its sensitivity to faults. These parameters are summarized in Table 5.1.

Table 5.1: Summary and description of each algorithm sensitivity parameter

Algorithm	Parameter	Range	Name
NSET+SPRT	α	0.005 – 0.2	False Alarm Probability
	β	0.005 – 0.2	Missed Alarm Probability
	m	1 – 6	System Disturbance Magnitude
	V	1 – 6	Variation Factor
Adaptive Threshold (AFFT)	K_{thr}	2 – 5	Threshold Multiplier
Order Analysis (OFFT)	K_{thr}	3.5 – 5.5	Threshold Multiplier

For this study, the OFFT's requirement of more than three data points outside of the threshold to log a fault is removed, to allow for the most accurate comparison between algorithms. To select the ideal value of each parameter, every possible combination of parameters is processed as an individual model. For each model, several measures to characterize the performance are defined. These measures are defined by ref. [20]. This approach treats each algorithm as a classification algorithm; that is, each algorithm can classify data into two classes: normal condition (NO), or a negative detection, and fault condition (FA), or a positive detection. The classification may have four outcomes:

1. Data is operating under NO and is classified as NO. (TN)
2. Data is operating under FA and is classified as FA. (TP)
3. Data is operating under NO and is classified as FA. (FP)
4. Data is operating under FA and is classified as NO. (FN)

These outcomes are also represented by the confusion matrix shown in Figure 5.2.

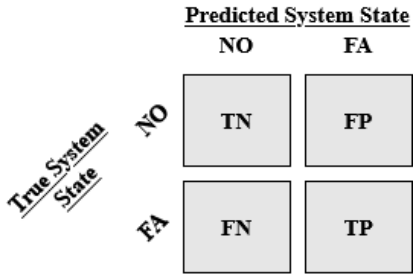


Figure 5.2: Confusion matrix of possible model outputs

The true positive rate, also known as recall, is the ratio of the number of positive detections TP to the total number of detections made while operating under the fault condition.

$$TP\ rate = Recall = \frac{TP}{TP + FN} \tag{5.2.a}$$

The precision, or confidence, is the ratio of the number of positive detections TP to the total number of positive detections.

$$Precision = \frac{TP}{TP + FP} \tag{5.2.b}$$

These measures are combined into a single score, known as the F-measure.

$$F - measure = 2\ \frac{Precision * Recall}{Precision + Recall} \tag{5.2.c}$$

The closer the F-measure is to one, the better a model's performance.

The final measure used is the false positive rate, or the ratio between false positives FP and the total number of true positives TP.

$$FP\ rate = \frac{FP}{FP + TN} \tag{5.2.d}$$

The false positive rate and total positive rate of each model may be visually described by a ROC chart. This chart is a 2D plot with each model's false positive rate on the X-axis, and its true positive rate on the Y-axis. Three major points describe a model's performance on the chart [20]:

a. (0,0): The model never classifies data into the positive/faulty state
b. (1,1): The model always classifies data into the positive/faulty state
c. (0,1): The model perfectly classifies data as faulty or healthy

Point c thus represents the "ideal" algorithm – it always classifies faulty data correctly, and never incorrectly raises an alarm with faulty data. However, for the application of algorithms in this thesis, reaching point c is not truly attainable. Practically, the vibration signal often drops for short periods, causing a faulty operating state to output vibrational data that appears healthy. This is pronounced when the rotor speed falls below 120 RPM; in this range, the vibrations induced by faulty operation are not enough to ever trigger any algorithm's alarm. To minimize this effect and allow for a more accurate representation of algorithm performance at operational speed, only operating data with a rotor speed > 120RPM is considered when calculating the measures described above. A ROC chart for all three algorithms is shown in Figure 5.3.

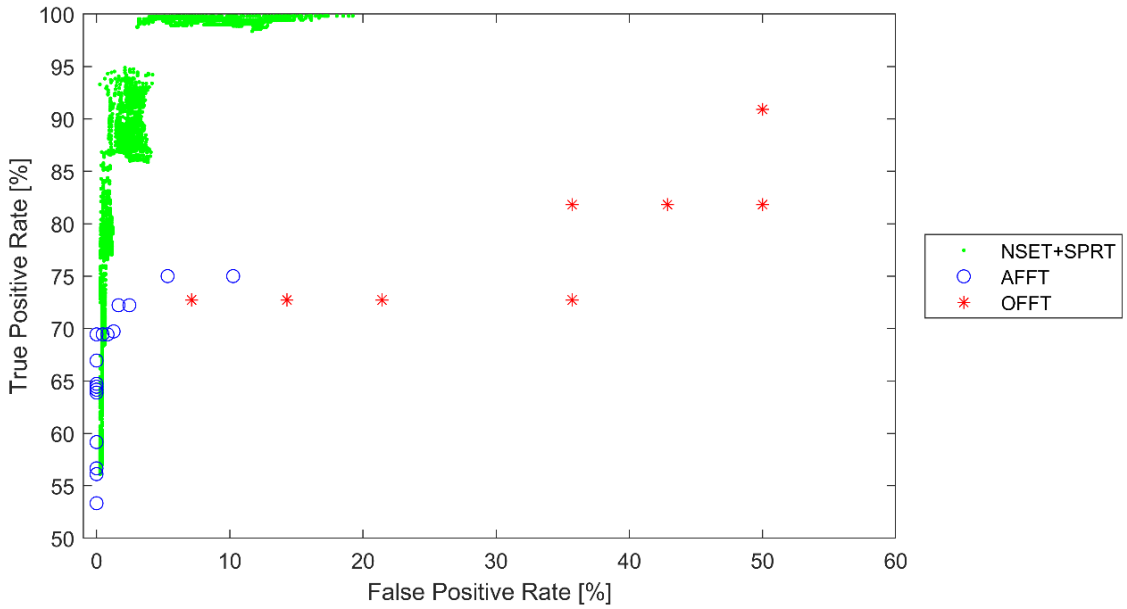


Figure 5.3: ROC chart showing performance for all combinations of parameters for each algorithm

Algorithm tuning parameters are chosen such that $FP\,rate$ is minimized and $TP\,rate$ is maximized, with the minimization of $FP\,rate$ being a much more important goal. The NSET+SPRT is much more tunable than the two FFT-based algorithms; because of this, many more possible models are shown. The parameters selected for each algorithm are shown in Table 5.2. Each algorithm's performance parameters are logged as well. One final measure is noted: the computation time necessary for a single run of the cross-validation study, wherein only the selected

parameters are studied. This is an important measure, as the end goal of these algorithms is live integration on a microcontroller-based monitoring system.

Table 5.2: Selected algorithm parameters and relevant performance data.

Algorithm	Parameter [#]	Selected Value [#]	FP_{rate} [%]	TP_{rate} [%]	F-measure [#]	Computation Time [s]
NSET+SPRT	m	2	0.25	93.3	0.964	25.82
	V	1				
	α	0.005				
	β	0.010				
AFFT	K_{thr}	2	0	69.44	0.820	11.04
OFFT	K_{thr}	5.5	7.14	72.73	0.800	78.08

All algorithms had a significant true positive rate and a very small false positive rate. The false positive rate for the NSET+SPRT was unable to be reduced to zero for any combination of parameters. Indeed, the 0.25% false positive result shown results from a single false positive within several of the 10 runs of the cross-validation study. Therefore, live implementation of this algorithm should not be allowed to shut down turbine operation without several positive detections within a predefined period. The OFFT was by far the most expensive algorithm computationally and the worst performing in terms of its false positive rate. The algorithm's speed may likely be optimized; however, owing to its poor performance, it is not recommended for implementation over the AFFT. However, the time required for the OFFT's synchronous sampling may be significantly improved by adopting an analog-based sampling method rather than the digital method discussed in section 4.3.

As discussed in section 1.4, the ideal condition monitoring algorithm must adhere to several characteristics. It must have a high correct alarm rate in detecting blade damage. Even more important, though, is to not raise false alarms during normal operation. It is clear, then, that the optimal solution is a joint monitoring approach that utilizes the high correct alarm rate of the

NSET+SPRT algorithm and eliminates false alarms as the AFFT does. This joint system will be designed for and implemented on the CPWT in chapter 6.

5.3 Study Limitations

The cross-validation study discussed contains some limitations which must be considered for applying these results to any data. First, the data was both trained by and tested on data with minimal rotor speed changes. For all previous applications of the turbine, including performance testing (see Cunningham's work, [10]) this is true. However, future operation may require running the turbine under large changes in speed, such as using tip-speed ratio control in conditions where the wind speed is changing quickly. These conditions, where both the wind speed and rotor speed quickly change, will likely lead to a higher chance of a false alarm.

Also, the effect of high winds on the algorithms was not thoroughly investigated, because a high windspeed control method does not yet exist – the current control methods result in the rotor speed becoming uncontrollable with wind speeds higher than ~20MPH. High winds will likely lead to the CMS raising alarms during healthy operation, especially for the NSET+SPRT algorithm. For this reason, implementation of the algorithms should include logic-based filtering that does not allow the algorithm to make fault determinations with wind speeds outside the learned range.

Chapter 6

CAL POLY WIND TURBINE IMPLEMENTATION

Transitioning the developed algorithms from a post-processing MATLAB environment to live integration on the wind turbine requires significant development. As discussed in chapter 2, the LifeLine collects data via a MicroPython microcontroller and transmits this information to a Raspberry Pi. The first iteration of the LifeLine CMS will be developed for the Raspberry Pi to simplify integration. Each algorithm to be implemented must be completely rebuilt in Python. The completion of this task requires three steps:

1. Deciding on the final CMS structure
2. Training the CMS with a healthy rotor vibrational response
3. Integrating the CMS in the Raspberry Pi's monitoring software

This chapter outlines each of these steps.

6.1 CMS Structure

Before any development may be completed, one must decide on which algorithm to implement. As discussed in chapter 5, both the NSET+SPRT and AFFT algorithms are very powerful. These algorithms both correctly raise true alarms when monitoring faulty data and nearly zero false alarms when monitoring healthy data. In addition, each one allows for a slightly different understanding of the turbine's health. The NSET+SPRT is useful as a *monitoring* algorithm that notes potential deviations from the normal operating state. Mass imbalances are only one of the potential issues caught; this algorithm may also identify broken sensors, failures in the automatic yaw control system, and even long-term reductions in power output due to blade surface damage. Additionally, it may identify unknown faults not appearing at a specific frequency, which the AFFT cannot detect. The AFFT, on the other hand, serves the purpose of *diagnosing* specific faults. Peaks in vibrations at 1X the rotor speed strongly indicate a mass unbalance is present, while peaks at 3X the rotor speed theoretically indicate an aerodynamic imbalance. Barszcz makes the distinction

between these two types of algorithms in ref. [36]. *Monitoring* serves as a low-level task meant to protect the machine from damage; in severe enough cases, threshold violations may even shut the machine down. *Diagnostics*, on the other hand, focus on early detection of faults and often simply alert operators to the fault presence.

Therefore, in the case of attempting to choose between them, the best scenario may be to not choose at all. In a combined implementation, the AFFT will note that a specific type of damage may occur, and the NSET+SPRT will determine if this damage is significant enough to raise an alarm. This integration is shown in Figure 6.1. As shown, the current implementation requires a fault determination from both the NSET+SPRT *and* the AFFT to raise an alarm. The reasoning here is to make the monitoring system more resistant to false positives, such as from large gusts of wind (which might trigger an NSET+SPRT alarm) or large changes in rotor speed (which might trigger an AFFT alarm). Five consecutive alarms from fault detections from either algorithm also send an alarm, which serves to significantly increase the CMS accuracy without resulting in a higher false alarm rate.

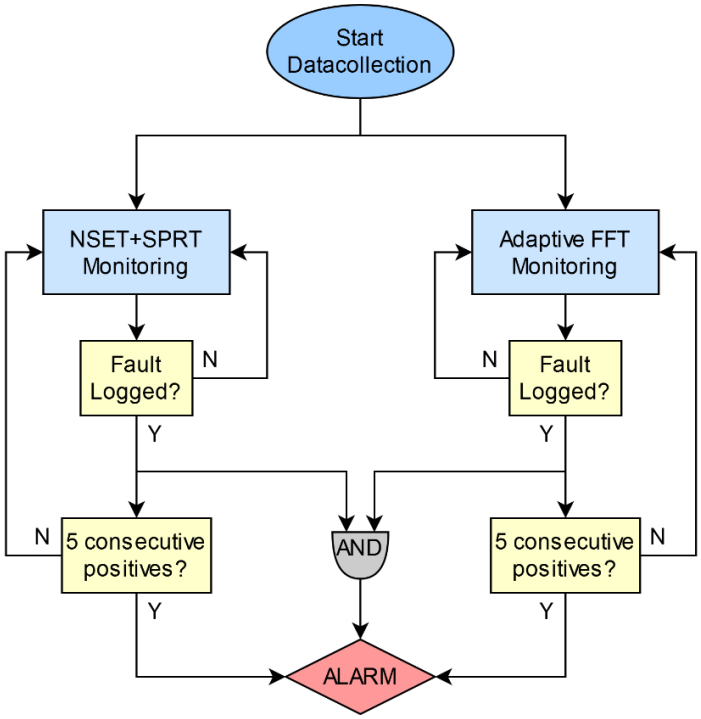


Figure 6.1: Flowchart of NSET+SPRT and AFFT integration on the CPWT

Another consequence of this joint method is that each algorithm may also be made more sensitive than if they were implemented alone. The additional logic of a second algorithm prevents false positives yet yields a significantly higher rate of correct fault detections.

6.2 CMS Training

Before either the NSET+SPRT or AFFT may be used to monitor turbine operation, they must learn the tower vibration response to normal operating conditions. In particular, the NSET+SPRT requires the formed memory matrix $\boldsymbol{D}$, and the AFFT requires unique frequency-based thresholds for each bin of rotor speed. Both quantities are formed, saved in text files, and loaded into the LifeLine logging software before monitoring begins. It is in this training phase that several opportunities to reduce the computation requirements of the algorithms appear. This is important, as it will minimize the possibility of the algorithms interfering with data acquisition or turbine control. For the NSET+SPRT, the quantity $\left(\boldsymbol{D}^{\mathrm{T}} \otimes \boldsymbol{D}\right)^{-1}$, used in equation (4.1.f), is also calculated and saved to a text file. Also, for the SPRT test, the simplified SPRT index calculation in (4.1.n) is used. With these performance-saving methods, the NSET+SPRT takes roughly 18ms to compute for every 10 seconds of data collected. As the AFFT is much faster computationally, it requires few changes for implementation. With this implementation, generating the NSET memory matrix and the AFFT thresholds outside the fault detection code is necessary. To assist in this, a MATLAB application was developed to generate fault detection text files. This tool is shown in Figure 6.2.

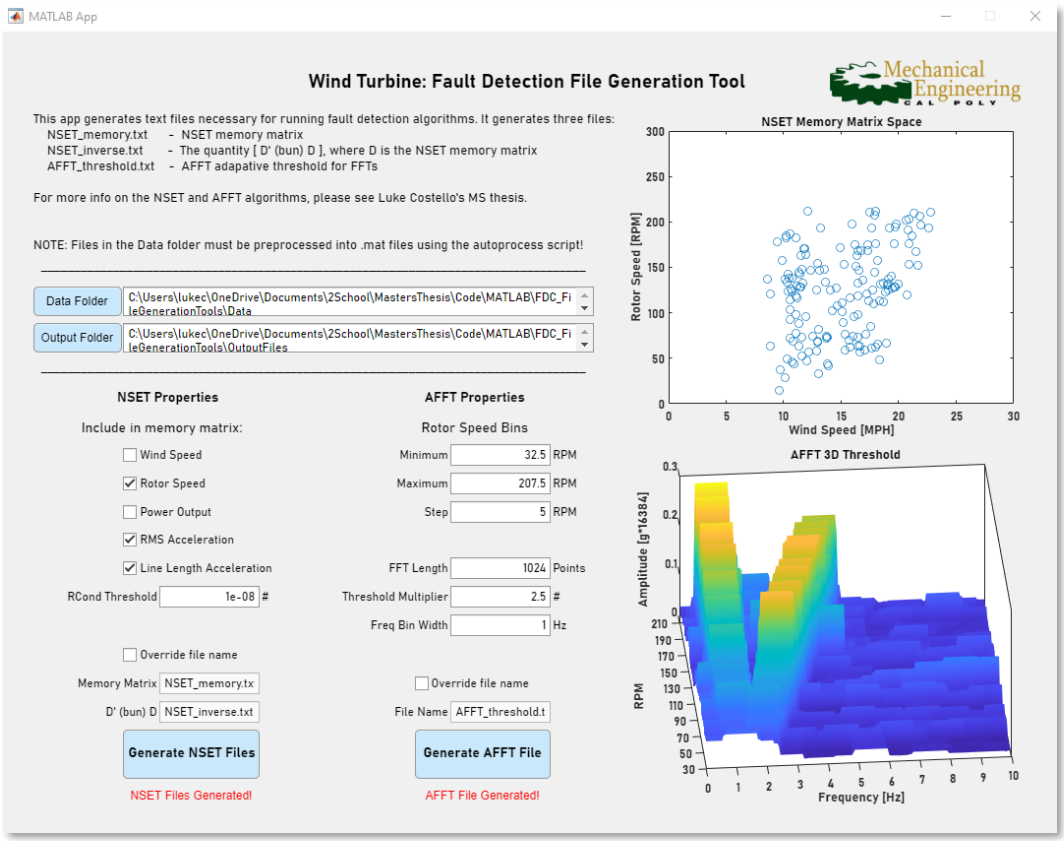


Figure 6.2: File Generation Tool GUI

This tool allows for setting many of the parameters used to form the text files for algorithm training.

6.3 CMS Implementation

The combined approach to monitoring was developed for the CPWT's Raspberry Pi through the development of three classes of functions. An overview of each class is shown in Table 6.1.

Table 6.1: List of the three Python-based classes of functions used for the Lifeline CMS

Algorithm / Class Name	Input	Output
NSET	Processed Data	Residual
SPRT	Residual	Fault Decision
AFFT	Processed Data	Fault Decision

The full code of these three classes may be found in I. Python Implementation Code. The primary goal of these three files is to simplify the implementation of each algorithm in a full

monitoring software. Each class requires three major steps: initializing an object of the class for use, sending data through this object, and interpreting the object's outputs. These three algorithms were included in the GUI currently being developed by Ryan Zhan [6]. See Figure 6.3 for the plotted outputs of these algorithms as they appear in the GUI, and Ryan Zhan's thesis for more information regarding the GUI design.

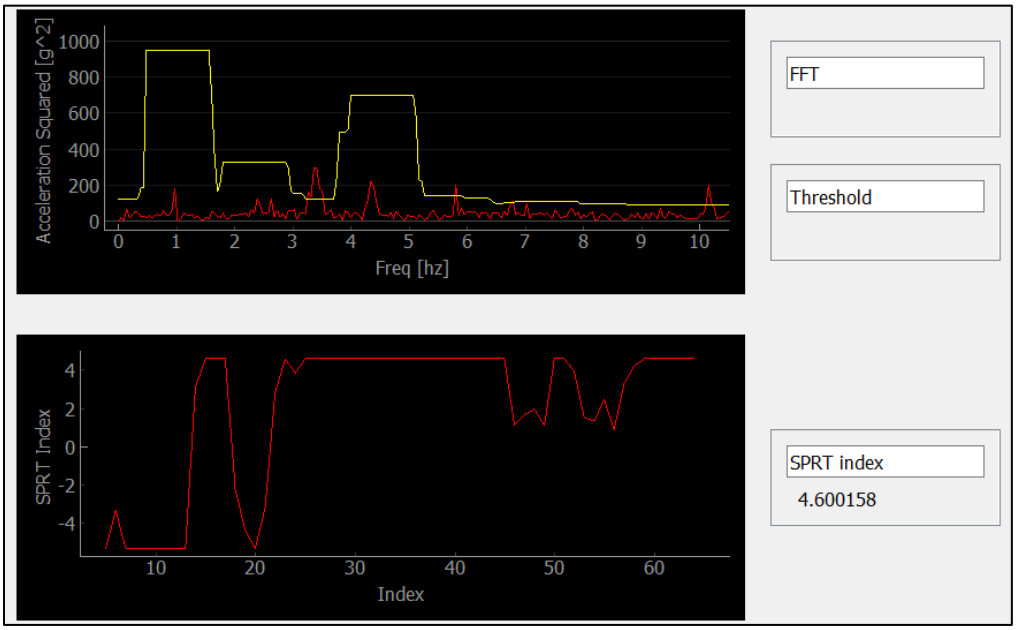


Figure 6.3: Plots of AFFT (top) and NSET+SPRT (bottom) as they appear in the LifeLine GUI while operating on faulty data

Once implemented on the LifeLine GUI, the complete CMS was tested on the turbine with a healthy rotor state to ensure that no false positives were present. In addition, the performance of the implemented CMS was again studied in a cross-validation study like that executed in chapter 5; the implemented version was fed collected data with the same logic as a live integration. Again, the detection parameters were tuned to maximize the correct detection rate and eliminate false positives. The final parameters are shown in Table 6.2.

Table 6.2: Final parameters for each algorithm, and associated CMS performance

Algorithm [#]	**Parameter** [#]	**Value** [#]	**True Positive Rate** [%]	**False Positive Rate** [%]	**F-score** [%]	**Accuracy** [%]
NSET+SPRT	m	2	90.2	0	0.945	95.4
	α	0.005				
	β	0.01				
AFFT	K_{thr}	1				

Ultimately, the final version of the CMS for the CPWT had both the high accuracy rate of the NSET+SPRT and zero false positives from the AFFT. Therefore, the finished version is robust enough for deployment on the CPWT in real-time – making the LifeLine capable of detecting large-scale blade damage and bringing the CPWT one step closer to autonomous operation.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In review, the major goal of the present work – to build, validate and implement a condition monitoring algorithm on the Cal Poly Wind Turbine – was successful. A working version of the joint monitoring solution, utilizing both the NSET+SPRT and AFFT algorithms, was implemented on the Raspberry Pi computer. A cross-validation study showed the final implementation had a 95.4% accuracy rate when classifying healthy and intentionally imbalanced data. Furthermore, post-implementation field testing showed the monitoring system raised no false positives during normal operation – a critical requirement for the final algorithm.

Field testing also yielded several major results but raised unanswered questions concerning the tower's vibrational response to loading. In time-series analyses, tower vibrations peak at the tower bending modes, with the bending modes mostly consistent with theoretical models of the tower. Of the time-series measures considered, these results also established the Line Length and RMS parameters as the best indicators of blade damage when monitoring tower vibrations. Frequency analyses, however, give results that both agree and disagree with established theory. Chief among these is the presence of peaks in the frequency spectrum that do not correspond to tower bending modes or multiples of rotating frequency (specifically, peaks in Figure 3.8 – Figure 3.10 at 1.1Hz and 1P + 1.1Hz). These peaks likely appear from unconsidered dynamic effects or sensor sampling issues; however, the cause is yet unknown. In addition, imbalance testing with a 200g mass caused the imbalanced blade to bend out of the rotor plane, and gave rise to a significant 3P frequency of vibrations. This was inconsistent with the theory of aerodynamic imbalances presented in section 1.2. It is theorized that the blade bending out of plane generated turbulent eddies, causing rotational sampling of turbulence – which does appear at the 3P frequency. However, this theory is unconfirmed. For the present work, it was simply enough to know that these effects occurred; they

could be ignored by the fault detection algorithms, leading to a robust condition monitoring system. However, there is ample opportunity for future testing to further investigate these effects.

For the continued growth of the LifeLine fault monitoring system, there are several areas in which future research might focus. There are several possible improvement paths, including:

1. Detecting damage to additional components
2. Improving detection of blade damage
3. Sensor validation
4. Better LifeLine integration

There are several other components on any wind turbine that fail often enough to justify monitoring them via a CMS like the one developed here. They include other rotor components, such as bearings and a gearbox (if present), the generator, any active yaw or blade pitch control system, and the tower structure. Monitoring each of these will require a unique combination of sensors and processing algorithms.

The blade damage detection methods discussed may also be improved. First, a method to directly monitor the blade structure, such as those discussed in ref. [17], will allow for earlier detection of damage than the methods described here. Improvements may also be made to the signal processing algorithms. The methods used in the present work all use statistical or absolute thresholds to distinguish between healthy and faulty operation. Although decently successful, these methods sometimes contain false positives in the event of extreme turbine operation. Therefore, a future thesis might investigate logic-based classification algorithms to decide on the presence of a fault. These range from simpler implementations, such as K-nearest-neighbors, to more complex topics such as fuzzy-logic classification. Another growing topic of consideration is the use of artificial intelligence (AI) based methods for classifying data.

Another recommended area of continued study is the addition of sensor validation. The end goal for the LifeLine is a modular system able to be applied to other wind turbines. Therefore, the LifeLine sensors must be resistant to harsh operating conditions, including electrical issues, sudden

load changes, and other mechanical problems. This may be ensured via a signal validation process. According to Barszcz [36], "the goal of signal validation is making a decision whether the acquired signal can be used for subsequent data analysis." Such a sensor validation involves continuously checking the sensor readings and ensuring no sensor faults have occurred. At this point, the LifeLine is a continuously changing system, so implementation of these sensor validation checks does not yet make sense. Once the LifeLine system is less prone to change, however, signal validation will be an important component of the LifeLine system before it is ready to be implemented on additional turbines. For additional discussion on this topic, see chapter 4 in ref. [36]. Incidentally, the NSET+SPRT was originally developed to detect sensor issues (specifically, fouling in feedwater flow sensors) – therefore, the extension of this algorithm to all sensors is a simple first step for a sensor validation process.

The final important change to the CPWT's current implementation is developing the ability for the LifeLine to notify the operator, or even shut down the turbine, in the event of a fault. This change will allow for the CPWT condition monitoring system to remotely monitor the turbine and open the door for research opportunities that require the long-term operation of the CPWT without any operator present. Research that requires long-term autonomous operation includes efforts to certify the CPWT under IEC standards, which is an area of ongoing study.

REFERENCES

[1] C. A. Walford, “Wind Turbine Reliability: Understanding and Minimizing Wind Turbine Operation and Maintenance Costs,” Sandia National Laboratories, Mar. 2006.

[2] A. Kusiak, W. Li, “The Prediction and Diagnosis of Wind Turbine Faults”, in *Elsevier Renewable Energy*, *36*(1), 16-23, 2011.

[3] C. B. Martínez-Anido, B.M. Hodge, “Impact of Utility-Scale Distributed Wind on Transmission-Level System Operations,” National Renewable Energy Laboratory, Golden, CO, TP-5D00-61824, 2014.

[4] “Small and Medium Wind Strategy: The current and future potential of the sub-500kW wind industry in the UK,” renewableUK, London, UK. [Online]. Available: https://cdn.ymaws.com/www.renewableuk.com/resource/resmgr/Docs/small_medium_wind_strategy_r.pdf

[5] “What is Distributed Wind?” Distributed Wind Energy Association. https://distributedwind.org/home/learn-about-distributed-wind/what-is-distributed-wind/

[6] R. Zhan, “Novel Software and Hardware for Wind Turbine Health Monitoring,” M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2021.

[7] R. Takatsuka, “Development of a Model and Imbalance Detection System for the Cal Poly Wind Turbine,” M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2019.

[8] Md A. S. Shohag, E. C. Hammel, D. O. Olawale, O. I. Okoli, “Damage mitigation techniques in wind turbine blades: A review,” *Wind Engineering,* vol. 41, no. 3, pp. 185-210, June 2017.

[9] A. C. Garolera, S. F. Madsen, M. Nissim, J. D. Myers, J Holboell, “Lightning Damage to Wind Turbine Blades From Wind Farms in the U.S,” *IEE Trans. on Power Delivery,* vol. 31, no. 3, pp. 1043-1049, June 2016.

[10] J. Cunningham, “Field Testing the Effects of Low Reynolds Number on the Power Performance of the Cal Poly Wind Power Research Center Small Wind Turbine,” M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2020.

[11] W. Qiao, “Recovery Act: Online Nonintrusive Condition Monitoring and Fault Detection for Wind Turbines,” University of Nebraska – Lincoln, NE, USA, Award no. DE-EE0001366, 2012.

[12] C. G. Anderson, *Wind Turbines: Theory and Practice*, Cambridge, UK: Cambridge University Press, 2020.

[13] S. Sheng, “Wind turbine Gearbox Condition Monitoring Round Robin Study – Vibration Analysis,” National Renewable Energy Laboratory, Jul. 2012.

[14] A. Lincoln, “What is Synchronous (Angular) Sampling?” Prosig, https://blog.prosig.com/2011/11/09/what-is-synchronous-angular-sampling/

[15] M. R. Shahriar, “Speed-based Diagnostics of Aerodynamic and Mass Imbalance in Large Wind Turbines,” presented at the IEEE Int. Conf. AIM, Busan, Korea, July 7-11, 2015, pp. 796-801.

[16] M. Roemer, *Private Conversation with P. Lemieux*, Sikorsky Helicopter, 2018.

[17] Y. Du, S. Zhou, X. Jing, Y. Peng, H. Wu, N. Kwok, “Damage detection techniques for wind turbine blades: A review,” in *Elsevier Mech. Systems & Sig. Processing*, China, 2019.

[18] T. Gwon, “Structural Analyses of Wind Turbine Tower for 3 kW Horizontal-Axis Wind Turbine,” M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2011.

[19] G. R. Katsanis, “Transient Small Wind Turbine Tower Structural Analysis with Coupled Rotor Dynamic Interaction,”, M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2013.

[20] H. Ahmed, A. K. Nandi, “Time Domain Analysis” in *Condition Monitoring with Vibration Signals*, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2020, ch. 3, sec. 3.2, pp. 35-41.

[21] M. M. Rahman, “Online Unbalanced Rotor Fault Detection of an IM Drive Based on Both Time and Frequency Domain Analyses,” *IEEE Trans. Industry Applications*, vol. 53, no. 4, pp. 4087 – 4096, Jul. 2017.

[22] D. Simon, “Static Balancing of the Cal Poly Wind Turbine Rotor,” M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2012.

[23] “von Kármán wind turbulence model,” Wikipedia. https://en.wikipedia.org/wiki/Von_K%C3%A1rm%C3%A1n_wind_turbulence_model

[24] J. Chen, R. J. Patton, “Basic Principles of Model-Based Fault Diagnosis,” in *Robust Model-Based Fault Diagnosis for Dynamic Systems*, New York, NY, USA: Springer Science+Business Media, 1999, ch. 2, pp. 19-64.

[25] F. K. Bockhorst, K. C. Gross, K. P. Herzog, and S. W. Wegerich, “MSET modeling of crystal river-3 venturi flow meters,” in *Proc. Int. Conf. Nuclear Engineering*, San Diego, CA, 1998.

[26] P. Guo, D. Infield, X. Yang, “Wind Turbine Generator Condition-Monitoring Using Temperature Trend Analysis,” in *IEEE Trans. Sustain. Energy*, 2012.

[27] P. Guo, D. Infield, “Wind Turbine Tower Vibration Modeling and Monitoring by the Nonlinear State Estimation Technique (NSET),” in *Energies*, 2012.

[28] R. M. Singer, K. C. Gross, K. P. Herzog, R. W. King, and S. Wegerich, “Model-based Nuclear Power Plant Monitoring and Fault Detection: Theoretical Foundations,” in *Proc. 9th Int. Conf. Intelligent Systems Applications to Power System,* Seoul, Korea, Jul. 1997.

[29] S. W. Wegerich, “Similarity Based Modeling of Time Synchronous Averaged Vibration Signals for Machinery Health Monitoring,” in *IEEE Aerospace Conference Proceedings*, Lisle, IL, 2004.

[30] “Sequential probability ratio test,” Wikipedia. https://en.wikipedia.org/wiki/Sequential_probability_ratio_test

[31] S. Cheng, K. Tom, L. Thomas, M. Pecht, “A Wireless Sensor System for Prognostics and Health Management,” in *IEEE Sensors Journal*, vol. 10, no. 4, pp. 856-862, Apr. 2010.

[32] T. J. Harrison, “The Sequential Probability Ratio Test (SPRT) in Feature Extraction and Expert Systems in Nuclear Material Management,” M.S. thesis, Dept. Nuclear Eng., University of Tennessee, Knoxville, TN, 2004.

[33] R. Isermann, “Supervision, fault-detection and diagnosis methods – a short introduction,” in *Fault-Diagnosis Applications*, New York, NY, USA: Springer Heidelberg Dordrecht, 2011, ch. 2, pp. 11-45.

[34] P. Caselitz, J. Giebhardt, “Rotor Condition Monitoring for Improved Operational Safety of Offshore Wind Energy Converters,” in *J. Solar Eng*, vol. 127, pp. 253-261, May 2005.

[35] W. Bartelmus, R. Zimroz, “A new feature for monitoring the condition of gearboxes in non-stationary operation conditions,” in *Mech. Systems, Sig. Processing*, vol. 23, no. 5, pp. 1528-1534, Jul. 2009.

[36] T. Barszcz, “Load-Susceptibility Characteristics,” in *Vibration-Based Condition Monitoring of Wind Turbines*, in Cham, Swizterland: Springer Nature, 2019, ch. 5, sec. 1, pp. 149-156, 2019.

[37] R. Zimroz, W. Bartelmus, T. Barszcz, J. Urbanek, “Diagnostics of bearings in presence of strong operation conditions non-stationarity-A procedure of load-dependent features processing with application to wind turbine bearings,” *Mech. Syst. Sig. Process*, vol. 46, no. 1, May 2014.

[38] A. Ghasemi, S. Zahediasl, “Normality Tests for Statistical Analysis: A Guide for Non-Statisticians,” in *Int J Endocrinol Metab*, 2012, 10(2),:486-489.

[39] R. Sandret, “Design, implementation, and testing of a control system for a small, off-grid wind turbine,” M.S. thesis, Dept. Mech. Eng., Cal Poly, San Luis Obispo, CA, 2012.

APPENDICES

A. Approximating the Matrix Condition Number

The algorithms developed to build a memory matrix for the Nonlinear State Estimation Technique rely on approximating the condition number of a matrix. This attachment compares the time saved by this approximation. Calculating the precise condition number of a matrix requires computing the singular values of a matrix, $\boldsymbol{S}$. For an m-by-m matrix $\boldsymbol{C}$, $\boldsymbol{S}$ will be an m-by-1 vector. Then, the condition number Λ is the ratio between the maximum and minimum singular values:

$$\Lambda = \frac{\max(\boldsymbol{S})}{\min(\boldsymbol{S})} \tag{A.1}$$

In comparison, Λ may be approximated by computing the 1-norm condition number:

$$\tilde{\Lambda} = \max\left(C_{ij} n_i\right) * \max\left({C^{-1}}_{ij} n_i\right) \tag{A.2}$$

The algorithm developed does not use the condition number beyond a simple threshold; therefore, exact computation of the exact condition number is not necessary. To quantify the difference in time between the exact and approximate methods, a MATLAB script finds the elapsed time for the calculation of Λ and $\tilde{\Lambda}$ given progressive steps in a n-by-9 matrix. This is taken 10,000 times, and the results averaged; this is shown in Figure A.. The shown value is given by $\% = (Time\ to\ compute\ \tilde{\Lambda}) \,/\, (Time\ to\ compute\ \Lambda)$. As shown, the time required for the approximation asymptotes at roughly 20% of the full calculation time. Variations largely occur due to changes in computer processing time; repeating these calculations does not yield the same peaks. However, the general trend is identical.

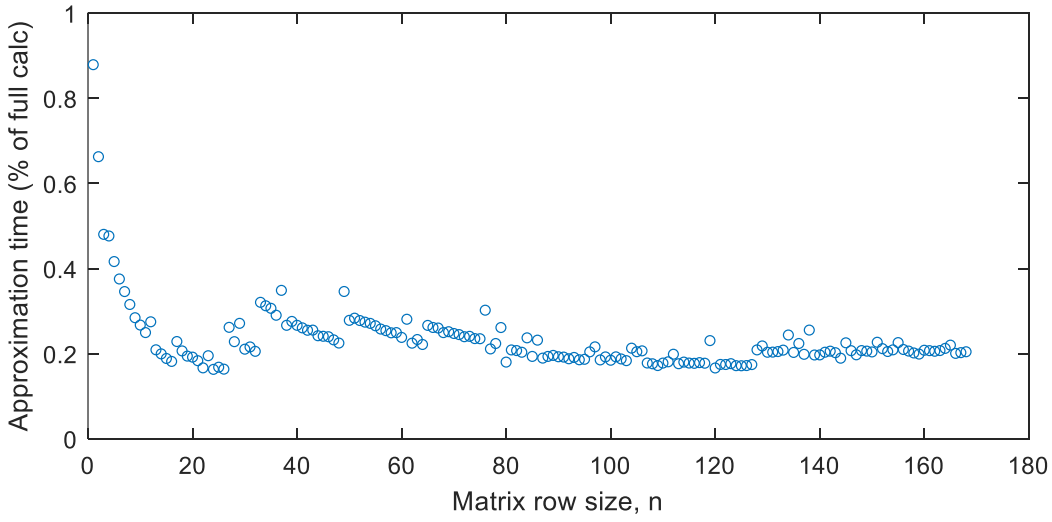


Figure A.1: Plot of the ratio between estimated and exact processing time vs matrix size.

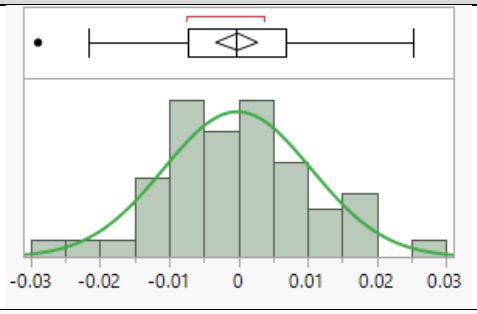
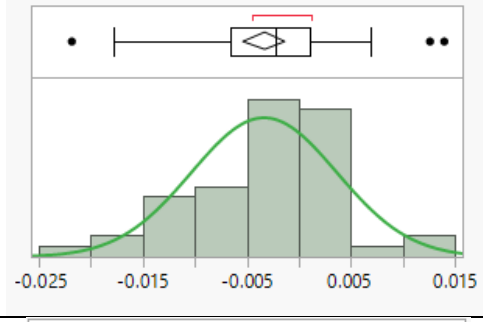
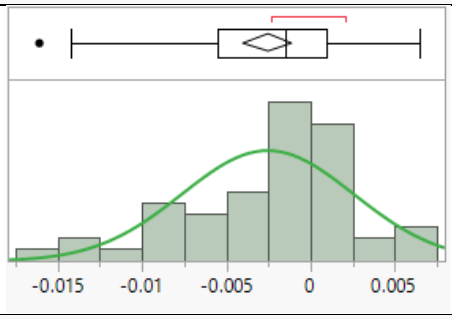
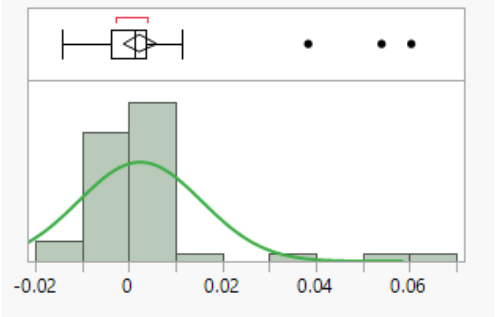
B. Testing the SPRT Normal Distribution Assumption

As discussed in section 4.1.4, the SPRT completed for the present work assumes that the residual computed by the NSET algorithm follows a normal distribution. A dataset may be said to be normally distributed if its probability density function may be described by the equation:

$$\Pr(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{B.1}$$

The normality of a dataset, or how well it adheres to a normal distribution, may be mathematically assessed via a Shapiro-Wilkes test. This test may be executed in the statistical software program JMP. To test this assumption, the entire set of residuals corresponding to healthy data from one iteration of the cross-validation study of chapter 5 is extracted from MATLAB and loaded into JMP. Once loaded, the Shapiro-Wilkes test is completed. This test is recommended for use with sample sizes no larger than 50; for this reason, the sample of 203 datapoints is divided into four sets and the test completed once for each set. The results of these tests are shown in Table B.. With a p-value of less than 0.05, the normality assumption is rejected. As shown, for the first two tests, the assumption is not rejected; however, it is rejected for the second two tests. This shows that as the rotor speed increases, the dataset moves farther from a normal distribution. This is likely caused by increased dynamic effects at higher speeds. Based on this test, the assumption that the dataset follows a normal distribution cannot always assumed to be true. Even with this incorrect assumption, however, the NSET+SPRT is shown to have a significant correct detection rate. Therefore, the SPRT will proceed with this assumption.

Table B.1: Testing results for a set of residuals from the NSET

Test [#]	Rotor Speed Range [RPM]	Histogram [#]	Test Statistic [#]	p-value [#]
1	30 – 75		0.991	0.9605
2	75 – 120		0.961	0.0934
3	120 – 165		0.9523	0.0426
4	165 – 210		0.640	<0.0001

C. Selection of Previous Student Works

Student(s)	Date	Purpose	Title
Jason Smith	9/2011	Site Assessment	California Polytechnic State University Wind Resource Assessment
Tom Gwon	8/2011	Tower Design	Structural Analyses of Wind Turbine Tower for 3 kW Horizontal Axis Wind Turbine
Bryan Edwards	9/2009	Blade Manufacturing	Composite Manufacturing of Small Wind Turbine Blades
Christopher Nosti	4/2009	Blade Performance Analysis	Performance Analysis and Life Prediction for Small Wind Turbine Blades: A Wood Laminate Case Study
Ka-Wah Li, Travis Robinson-Carter, Michael Julgebich	6/2009	Load Bank Design	Cal Poly Wind Turbine Off-Grid Load Bank and Emergency Speed Controller
Richard Sandret	6/2012	Control System Design	Design, implementation, and testing of a control system for a small, off-grid wind turbine
Derek Simon	8/2012	Blade Balancing	Static Balancing of the Cal Poly Wind Turbine Rotor
Kent Burnett	6/2012	Controller Design	A Proposed Control Solution for the Cal Poly Wind Energy Capture System
David Nevarez, Francisco Martinez, Alvaro Martinez	6/2008	Nacelle / Drivetrain Design	Nacelle Final Design Report
George Katsanis	5/2013	Rotor/Tower Modeling	Transient Small Wind Turbine Tower Structural Analysis With Coupled Rotor Dynamic Interaction
Dylan Perry	6/2015	Blade Modeling	Aerodynamic Design and Structural Analysis Procedure for Small Horizontal-Axis Wind Turbine Rotor Blade
John Cunningham	12/2020	Blade Characterization	Field Testing the Effects of Low Reynolds Number on the Power Performance of the Cal Poly Wind Power Research Center Small Wind Turbine

D. NSET Code

NOTE: All code is subject to change. See https://github.com/elceenor/LifeLine_FaultDetection for the latest version of all code presented herein. Code is included to allows readers to understand the developed algorithm's logical flow; copying into a code editor or IDE will likely introduce errors. As such, see the GitHub repository for a functional version of all programs.

1. prop.m

```
%Defines properties of the memory/measurement vectors to be memorized.
%s: List of columns to save
%n: Quantities to normalize each column
%d: Other data, including searching bins and thresholds
function [s,n,d] = prop()

    %Define array of column # to save
    % 1 = Time Column [s]
    % 2 = Wind Speed [MPH]
    % 3 = Rotor Speed [RPM]
    % 4 = Gen Voltage [VDC]
    % 5 = Gen Current [ADC]
    % 6 = Battery Voltage [VDC]
    % 7 = Nacelle angle [°]
    % 8 = Wind angle [°]
    % 9 = RMS Accel in rotor direction
    % 10= RMS Accel in transverse horz direction
    % 11= RMS Accel in transverse vert direction
    % 12= Line Length in rotor direction
    % 13= Line Length in transverse horz direction
    % 14= Line Length in transverse vert direction
    s = [2,3,4,5,9,10,11,12,13,14];

    pow = [4 5];

    %Define quantity to normalize each column # by
    n = [24,240,300,5,4*1024,4*1024,18*1024,5E5,5E5,5E5];

    %Define other properties for searching
    %findRPM  = divisions for RPM
    %deltR    = 0.5*distance between divisons for RPM
    %findWind = divisions for windspeed
    %deltW    = 0.5*distance between divisons for windspeed
    findRPM  = (30:10:180)/n(3);
    deltR = 0.5*(findRPM(2)-findRPM(1));
    findWind = (8:23)/n(2);
    deltW = 0.5*(findWind(2)-findWind(1));
    rcondThresh = 10E-9;


    d = {findRPM,deltR,findWind,deltW,rcondThresh,pow};

end
```

2. load_file.m

```
%/=====================================================================\
%|  PURPOSE: Loads MATLAB .mat files and processes them into a format |
%|           useful by the NSET+SPRT algorithms. Data files must be a |
%|           .mat file with the desired dataset variable "data"       |
%|---------------------------------------------------------------------|
%|  INPUTS:  fileName - The name of the file to be loaded, as a string|
%|---------------------------------------------------------------------|
%|  OUTPUTS: array    - The output array, ready for processing by NSET|
%|---------------------------------------------------------------------|
%|  Luke Costello, 9/26/2020                                           |
%\=====================================================================/

function [array] = load_file(fileName)
    global loud

    if loud
        fprintf('Loading file: %s\n',fileName)
    end
    clear data

    load(fileName)
    [sv,nrml,~] = prop();

    %Save only the sensors specified in prop()
    array = data(:,sv);
    array = array';

    %Normalize each sensor by values specified in prop()
    for i = 1:length(nrml)
        array(i,:)=array(i,:)/nrml(i);
    end

    %Compute power
    array(3,:) = array(3,:).*array(4,:);
    array(4,:) = [];
end
```

3. memorize.m

```
%/====================================================================\
%|   PURPOSE: Memorizes the input dataset 'data'. It is expected this |
%|            data adheres to NSET matrix convention, that is, each   |
%|            column is a vector of sensor values and each row is     |
%|            additional vectors in time.                             |
%|--------------------------------------------------------------------|
%|   INPUTS:  data - Input dataset to be memorized                    |
%|--------------------------------------------------------------------|
%|   OUTPUTS: mem  - Memorized matrix                                 |
%|--------------------------------------------------------------------|
%|   Luke Costello, 10/12/20                                          |
%\====================================================================/
function [mem] = memorize(data)
    global loud

    %Find extreme sensor data
    [mem1,data] = find_extremes(data);
    %Sort sensor data & form the largest memory matrix possible
    [mem2] = sort_step(data);

    %Combine sets and report
    mem = [mem1 mem2];
    val = rcond_e(mem);
    if loud
        fprintf('RCond # of formed memory matrix is: %2.2s\n',val);
    end
end
```

4. find_extremes.m

```
%/====================================================================\
%|  PURPOSE: Finds extreme sensor values for the sensors in position 2|
%|           and position 3.                                          |
%|--------------------------------------------------------------------|
%|  INPUTS:  array - Input dataset to be memorized                    |
%|--------------------------------------------------------------------|
%|  OUTPUTS: out   - Memorized matrix                                 |
%|           array - Returns the input array without the memorized    |
%|                   values so that later memorization algorithms do  |
%|                   not use them as well.                            |
%|--------------------------------------------------------------------|
%|   Luke Costello, 10/12/20                                          |
%\====================================================================/
function [out,array] = find_extremes(array)

    [~,i] = min(array(2,:));
```

```
    out = [array(:,i)];
    array(:,i) = [];

    [~,k] = min(array(3,:));
    out = [out array(:,k)];
    array(:,k) = [];

    [~,j] = max(array(2,:));
    out = [out array(:,j)];
    array(:,j) = [];

    [~,l] = max(array(3,:));
    out = [out array(:,l)];
    array(:,l) = [];

end
```

5. sort_step.m

```
%/=====================================================================\
%|   PURPOSE: Sorts sensor vectors into a memory matrix. This is done  |
%|            by determining the range of operating data for each      |
%|            sensor and dividing into 1/k steps. The vector with      |
%|            sensor measurement closest to each step is saved into    |
%|            the memory matrix. k is progressively increased (for     |
%|            each sensor) until a memory matrix is create where each  |
%|            vector is sufficiently unique that the Rcond # is        |
%|            greater than the threshold set by prop().                |
%|---------------------------------------------------------------------|
%|   INPUTS:  data - Input Sensor Data                                 |
%|---------------------------------------------------------------------|
%|   OUTPUTS: mem  - Formed memory matrix                              |
%|---------------------------------------------------------------------|
%|   Luke Costello, 10/12/20                                           |
%\=====================================================================/

function [mem] = sort_step(data)

    n = size(data,1);
    min_max = zeros(n,2);

    k_n = 0.01*ones(n,1);

    [~,~,d] = prop();
    rcondThresh = d{5};

    %Find minimum and maximum values for each sensor
    for i = 1:n
        min_max(i,1) = min(data(i,:));
        min_max(i,2) = max(data(i,:));
    end

    %Calc difference between min & max values
    diff = min_max(:,2) - min_max(:,1);

    %Start sorting data
```

```
    done = false;
    while ~done
        num_steps = 1./k_n;

        test_mem = [];
        test_data = data;
        rcond_mat = zeros(n,1);
        restart = false;
        %Loop over each sensor
        for i = 1:n
            test_mem_i = [];
            %Calculate step size for sensor i
            step_val = (diff(i)/num_steps(i));
            %Loop over each step
            for j = 1:num_steps(i)
                %Calculate value to search for
                search_val = min_max(i,1) + j*step_val;
                %Search for value
                [~,ind] = min(abs(test_data(i,:) - search_val));

                %fprintf('Sensor: %d || Search value: %4.2f || Index
chosen: %4.2f\n',i,search_val,ind)
                test_mem_i = [test_mem_i test_data(:,ind)];
                test_data(:,ind) = [];

                %pause

                if isempty(ind)
                    break
                end
            end

            if isempty(ind)
                k_n = k_n + 0.01;
                restart = true;
                break
            end

            rcond_mat(i) = rcond_e(test_mem_i);
            test_mem = [test_mem test_mem_i];
        end

        if restart
            continue
        end

        rcond_curr = rcond_e(test_mem);
        if rcond_curr > rcondThresh
            done = true;
        else
            [~,ind] = min(rcond_mat);
            k_n(ind) = k_n(ind) + 0.01;
            %fprintf('Rcond too small (%2.2s)! Increasing step size for
sensor %d and continuing...\n',rcond_curr,ind)
        end
    end
```

```
    mem = test_mem;
end
```

6. rcond_e.m

```
%/=========================================================================\
%|  PURPOSE: Calculates the RCond number (see L. Costello's Thesis        |
%|           report or MATLAB's rcond function documentation)             |
%|------------------------------------------------------------ -------------|
%|  INPUTS:   mem - Matrix to compute RCond # for                          |
%|------------------------------------------------- ---------------------|
%|  OUTPUTS:  out - RCond Number                                           |
%|-------------------------------------------------------------------------|
%|  Luke Costello, 8/28/2020                                               |
%\=========================================================================/
function [out] = rcond_e(mem)

    m = size(mem,2);
    Dt_D = zeros(m);

    for i = 1:m
        for j = 1:m
            Dt_D(i,j) = euclid(mem(:,i),mem(:,j));
        end
    end

    out = rcond(Dt_D'*Dt_D);
end
```

7. euclid.m

```
%/=========================================================================\
%|   PURPOSE: Calculates the Euclidean distance between 2 vectors          |
%|-------------------------------------------------------------------------|
%|   INPUTS:  vec1 - Vector 1                                              |
%|            vec2 - Vector 2                                              |
%|-------------------------------------------------------------------------|
%|   OUTPUTS: out  - Euclidean Distance                                    |
%|-------------------------------------------------------------------------|
%|   Luke Costello, 9/12/20                                                |
%\=========================================================================/
function [out] = euclid(vec1,vec2)

    len_vec1 = length(vec1);
    len_vec2 = length(vec2);

    s = 0;

    for i = 1:len_vec1
        s = s + (vec1(i) - vec2(i))^2;
    end
```

```
    out = sqrt(s);
end
```

8. estimate_sensors.m

```
%/=====================================================================\
%|   PURPOSE: Estimates the expected sensor data given a memory matrix|
%|---------------------------------------------------------------------|
%|   INPUTS:  data - Dataset to be estimated                           |
%|            mem  - Memory Matrix                                     |
%|---------------------------------------------------------------------|
%|   OUTPUTS: est  - Estimated sensor values                           |
%|---------------------------------------------------------------------|
%|   Luke Costello, 10/15/20                                           |
%\=====================================================================/
function [est] = estimate_sensors(data,mem)

    [sv,nrml,~] = prop();

    array = data;


    for n = 1:size(data,2)
        obs = array(:,n);
        [out] = weight(mem,obs);
        est(:,n) = mem*out;
    end

    index = 1:n;
end
```

9. weight.m

```
%/=====================================================================\
%|   PURPOSE: Computes the weighting vector for NSET given a memory    |
%|            matrix and a new observed sensor vector.                 |
%|---------------------------------------------------------------------|
%|   INPUTS:  mem - Memory Matrix                                      |
%|            obs - Observed sensor vector                             |
%|---------------------------------------------------------------------|
%|   OUTPUTS: out - Weighting Vector                                   |
%|---------------------------------------------------------------------|
%|   Luke Costello, 8/25/20                                            |
%\=====================================================================/
function [out,invers] = weight(mem,obs)

    %Calculate size of input matrices
    n = size(mem,1);
    m = size(mem,2);

    if size(obs,1) ~= n
```

```
        error('Observed vector is not the same length as memorized
vectors.')
    end

    %Preallocate memory for matrices
    Dt_D = zeros(m,m);

    Dt_X = zeros(m,1);

    %Compute Euclidian distance between each memory vector, as well as
each
    %memory vector and observation vector
    for i = 1:m
        for j = 1:m
            Dt_D(i,j) = euclid(mem(:,i),mem(:,j));
        end
        Dt_X(i) = euclid(mem(:,i),obs);
    end


    %Compute inv(Dt_D)*Dt_X

    invers = inv(Dt_D);
    out = Dt_D\Dt_X;
end
```

E. SPRT Code

1. hypothesis.m

```
%/=====================================================================\
%|  PURPOSE: Determines the parameters of the hypothesis test          |
%|---------------------------------------------------------------------|
%|  INPUTS:  H_j     - Hypothesis to test.                             |
%|                     1 == mu  = +M  sig = sig(trained_data)          |
%|                     2 == mu  = -M, sig = sig(trained_data)          |
%|                     3 == mu  =  0, sig = V*sig(trained_data)        |
%|                     4 == mu  =  0, sig = (1/V)*sig(trained_data)    |
%|           S       - SPRT Parameters                                 |
%|---------------------------------------------------------------------|
%|  OUTPUTS: mu_test  - The mean value of the alternative hypothesis   |
%|           sig_test - The standard deviation of the alternative      |
%|                      hypothesis                                     |
%|---------------------------------------------------------------------|
%|  Luke Costello, 10/6/2020                                           |
%\=====================================================================/

function [mu_test, sig_test] = hypothesis(H_j,S)


    %Extract SPRT parameters
    sig_tr = S(1);
    M      = S(2);
    V      = S(3);

    %Determine hypothesis to test
    if H_j == 1
        mu_test = M;
        sig_test = sig_tr;
    elseif H_j == 2
        mu_test = -M;
        sig_test = sig_tr;
    elseif H_j == 3
        mu_test = 0;
        sig_test = V*sig_tr;
    elseif H_j == 4
        mu_test = 0;
        sig_test = (1/V)*sig_tr;
    end

end
```

2. LR.m

```
%/====================================================================\
%|   PURPOSE: Calculates the new likelihood ratio of a sequence for a |
%|            new datapoint added to the sequence.                    |
%|--------------------------------------------------------------------|
%|   INPUTS:  x    - New datapoint to test                            |
%|            lk_0 - Likelihood ratio, without taking new datapoint   |
%|                   into account.                                    |
%|            j    - Hypothesis to test.                              |
%|                     1 == mu  = +M  sig = sig(trained_data)         |
%|                     2 == mu  = -M, sig = sig(trained_data)         |
%|                     3 == mu  =  0, sig = V*sig(trained_data)       |
%|                     4 == mu  =  0, sig = (1/V)*sig(trained_data)   |
%|            S    - Vector containing SPRT parameters                |
%|                 S[1] = sig(trained_data) || S[2] = M  ||  S[3] == V|
%|--------------------------------------------------------------------|
%|   OUTPUTS: lk_1 - Likelihood ratio after taking new datapoint into |
%|                   account.                                         |
%|--------------------------------------------------------------------|
%|   Luke Costello, 10/6/2020                                         |
%\====================================================================/


function [lk_1] = LR(x,lk_0,j,S)

%Extract SPRT parameters
sig_tr = S(1);
M      = S(2);
V      = S(3);

%Determine hypothesis to test
[mu_test, sig_test] = hypothesis(j,S);


H_0 = normal_prob(x,0,sig_tr);
H_j = normal_prob(x,mu_test,sig_test);

if 1 == 2
    fprintf('Datapoint: %2.2f || Mu_test: %2.2e || Sig_test:
%2.2e\n',x,mu_test,sig_test)
    fprintf('Null hypothesis probability: %2.4e || Alternative
hypothesis probability: %2.4e\n',H_0,H_j)
end

lk_i = H_j/H_0;

lk_1 = lk_0 * lk_i;
```

```
%/=====================================================================\
%|   PURPOSE: Checks the probability that a datapoint is from a normal|
%|            distribution.                                           |
%|--------------------------------------------------------------------|
%|  INPUTS:  x    - datapoint to be checked                           |
%|           mu   - mean value of distribution                        |
%|           sig  - standard devation of distribution                 |
%|--------------------------------------------------------------------|
%|  OUTPUTS: prob - probability of x residing in distribution         |
%|--------------------------------------------------------------------|
%|  Luke Costello, 10/6/2020                                          |
%\=====================================================================/

function [prob] = normal_prob(x,mu,sig)
    prob = (sig*sqrt(2*pi))^-1 * exp(-0.5 * ((x-mu)/sig)^2);

end
```

```
%/=====================================================================\
%|   PURPOSE: Computes a SPRT for new data, using learned memory      |
%|            matrix and new data.                                    |
%|--------------------------------------------------------------------|
%|   INPUTS:  H_j     - Hypothesis to test.                           |
%|                      1 == mu  = +M  sig = sig(trained_data)        |
%|                      2 == mu  = -M, sig = sig(trained_data)        |
%|                      3 == mu  =  0, sig = V*sig(trained_data)      |
%|                      4 == mu  =  0, sig = (1/V)*sig(trained_data)  |
%|--------------------------------------------------------------------|
%|   OUTPUTS: mu_test  - The mean value of the alternative hypothesis |
%|            sig_test - The standard deviation of the alternative    |
%|                       hypothesis                                   |
%|--------------------------------------------------------------------|
%|   Luke Costello, 10/6/2020                                         |
%\=====================================================================/


function [alarms,SPRT_sv,range] = test_data(X_n,S,rots)
    global loud

    %Extract SPRT data
    sig = S(1);
    M = S(2);
    V = S(3);
    alph   = S(4);
    beta   = S(5);

    num_hyp  = 4; %Number of hypotheses to test

    %Define testing parameters
    A = log(beta/(1-alph));
    B = log((1-beta)/alph);
    range = [A B];

    decision = zeros(num_hyp,1);
    lk_0 = ones(num_hyp,1);
    lk_i = ones(num_hyp,1);
    alarms = zeros(num_hyp,2);

    %Test new data against training data
    %Loop over each datapoint
    for i = 1:size(X_n,2)

        %Extract datapoint to test
        %X_i = X_n(test_row,i);
        X_i = X_n(i);


        %if rem(i,50) == 0
        %    fprintf('On datapoint %d\n',i)
        %end

        %Loop over each hypothesis to test

        for j = 1:num_hyp
```

```
        %Compute likelihood ratio for new datapoint
        lk_i(j) = LR(X_i,lk_0(j),j,S);
        if isnan(lk_i(j))
            lk_i(j) = lk_0(j);
        end
        lk_0(j) = lk_i(j);
        %Compute SPRT index
        SPRT_i = log(lk_i(j));

        %Compare SPRT index to boundaries
        if SPRT_i >= A && SPRT_i <= B
            decision(j) = 1;
        elseif SPRT_i < A
            decision(j) = 2;
            SPRT_i = A;
        elseif SPRT_i > B
            decision(j) = 3;
            SPRT_i = B;
        end

        SPRT_sv(j,i) = SPRT_i;

        if rots(i) > 120
            if decision(j) ~= 1

                if decision(j) == 2
                    alarms(j,1) = alarms(j,1) + 1;
                elseif decision(j) == 3
                    alarms(j,2) = alarms(j,2) + 1;
                end
                decision(j) = 1;
                lk_0(j) = 1;
            end
        end

        if 1 == 2
            figure(3);
            hold on
            subplot(1,num_hyp,j)
            xlim([A,B])
            ylim([0,size(X_n,2)])
            plot(SPRT_i,i,'k.')
            pause
        end

    end


end

if 1 == 2
    ind = 1:size(SPRT_sv,2);
    figure(3);
    ylabel('Number Datapoints [N]')
    xlabel('Detection Range [A,B]')
    subplot(1,4,1)
    hold on
```

```
        plot(SPRT_sv(1,:),ind,'k.')
        plot([A A],[0 ind(end)],'-r')
        plot([B B],[0 ind(end)],'-r')
        xlim([A-0.5,B+0.5])
        ylim([0,size(X_n,2)])

        subplot(1,4,2)
        hold on
        plot(SPRT_sv(2,:),ind,'k.')
        plot([A A],[0 ind(end)],'-r')
        plot([B B],[0 ind(end)],'-r')
        xlim([A-0.5,B+0.5])
        ylim([0,size(X_n,2)])

        subplot(1,4,3)
        hold on
        plot(SPRT_sv(3,:),ind,'k.')
        plot([A A],[0 ind(end)],'-r')
        plot([B B],[0 ind(end)],'-r')
        xlim([A-0.5,B+0.5])
        ylim([0,size(X_n,2)])

        subplot(1,4,4)
        hold on
        plot(SPRT_sv(4,:),ind,'k.')
        plot([A A],[0 ind(end)],'-r')
        plot([B B],[0 ind(end)],'-r')
        xlim([A-0.5,B+0.5])
        ylim([0,size(X_n,2)])
    end
    %{
    if loud
        fprintf('         # Alarms:\n')

        fprintf('H_0:      1     2      3     4\n      ')
        disp(alarms')
    end
    %}

end
```

F. Adaptive FFT Code

1. AFFT_prop.m

```
%Saves properties for the AFFT
function [K,R_SPD_bins,d] = AFFT_prop(K)

if ~exist('K','var')
    K = 1.5;
end

R_SPD_bins = [32.5:5:207.5];


%Sample frequency
Fs = 50;
%Number datapoints per FFT
N = 1024;
%Width of Frequency Bins
bin_width = 1;

d = [Fs,N,bin_width];
```

2. learn_thresh2.m

```
%/=========================================================================\
%|   PURPOSE: Learns an adaptive threshold using training data FFTs        |
%|-------------------------------------------------------------------------|
%|   INPUTS:       amps - Amplitude of most recent FFT                     |
%|                freqs - Frequencies corresponding to amplitudes of       |
%|                        FFT                                              |
%|                K_thr - Threshold multiplier                             |
%|              rots_ar - Array of rotor speeds, to sort newly formed      |
%|                        thresholds into bins                             |
%|            threshold - The currently formed threshold, so it can be     |
%|                        compared against the next FFT                    |
%|-------------------------------------------------------------------------|
%|   OUTPUTS: threshold - The latest formed threshold                      |
%|-------------------------------------------------------------------------|
%|   Luke Costello, 10/6/2020                                              |
%\=========================================================================/

function [threshold] =learn_thresh2(amps,freqs,K_thr,rots_ar,threshold)
global debug


%%Sort each FFT into bins based on rotorspeed
[~,r_bins,d] = AFFT_prop(K_thr);

bin_width = d(3);                   %Frequency width per threshold
frq_step = (freqs(2)-freqs(1));     %Step size per FFT datapoint [Hz]
num_steps = ceil(bin_width/frq_step);     %Steps per frequency width
bin [#]
num_1side = ceil(num_steps/2);      %Steps on one side per peak
```

```
amps_cell = cell(1,length(r_bins));

for i = 1:size(amps,2)
    [~,x] = min(abs(rots_ar(i) - r_bins));
    amps_cell{x} = [amps_cell{x} amps(:,i)];
end


%%Loop over each bin of rotorspeed
for i = 1:length(amps_cell)
    amps_lrn = amps_cell{i};
    amps_lrn = max(amps_lrn,[],2);
    if any(size(amps_lrn) == [0 0])
        continue
    end

    for j = 1:length(amps_lrn)
        low = j-num_1side;
        high = j+num_1side;
        if low < 1
            low = 1;
        elseif high > length(amps_lrn)
            high = length(amps_lrn);
        end

        lrns = amps_lrn(low:high);
        max_lrns = K_thr * max(lrns,[],1);
        max_all = max([max_lrns,threshold(j,i)]);
        threshold(j,i) = max_all;
    end

    if debug
        figure(1);
        clf
        hold on
        size(amps_lrn)
        plot(freqs,amps_lrn,'-k')
        plot(freqs,threshold(:,i),'-r')
        fill([freqs;25;-
25],[threshold(:,i);max(threshold(:,i))+0.04;max(threshold(:,i))+0.04],
'r','FaceAlpha','0.1','LineStyle','None')
        pause
    end
end
```

3. test_thresh2.m

```
%/====================================================================\
%|   PURPOSE: Tests the FFT of vibration data against an adaptive     |
%|            threshold set earlier in the program.                   |
%|--------------------------------------------------------------------|
%|   INPUTS:   thresh - Array of currently formed thresholds          |
%|              freqs - Array of frequencies corresponding to         |
%|                      threshold and FFT                             |
%|               amps - Amplitude of FFT to compare to threshold      |
%|            rots_ar - Array of rotor speeds of data being tested    |
%|             r_bins - Array of rotor speed bins to sort data into   |
%|               nums - A vector of the number of detections:         |
%|                      [Number of Tests, Number of Positives]        |
%|            frq_pos - A vector of positive frequencies at which a   |
%|                      fault has been detected.                      |
%|--------------------------------------------------------------------|
%|   OUTPUTS: nums    - A vector of the number of detections:         |
%|                      [Number of Tests, Number of Positives]        |
%|            frq_pos - Vector of the frequencies at which a positive |
%|                      detection occurs.                             |
%|--------------------------------------------------------------------|
%|   Luke Costello, 10/6/2020                                         |
%\====================================================================/
function [nums,frq_pos] =
test_thresh2(thresh,freqs,amps,rots_ar,r_bins,nums,frq_pos)
    global debug
    global show_faults

    num_tests = nums(1);
    num_pos = nums(2);

    %Sort FFT output into cells based on rotor speed
    amps_cell = cell(1,length(r_bins));

    for i = 1:size(amps,2)
        [~,x] = min(abs(rots_ar(i) - r_bins));
        amps_cell{x} = [amps_cell{x} amps(:,i)];
    end

    %Loop over each rotor speed
    for i = 1:length(amps_cell)
        amps_test = amps_cell{i};
        %Loop over each FFT at the current rotorspeed
        for k = 1:size(amps_test,2)

            amp_test = amps_test(:,k);
            thresh_test = thresh(:,i);
            test = find(amp_test>thresh_test);

            if debug
                figure(1);
                clf
                hold on
                plot(freqs,amp_test,'-k')
                plot(freqs,thresh(:,i),'-r')
```

```
                fill([freqs;25;-
25],[thresh(:,i);max(thresh(:,i))+0.04;max(thresh(:,i))+0.04],'r','Face
Alpha','0.1','LineStyle','None')
                pause
            end


            %If test isn't empty, than a fault has been detected
            if ~isempty(test)
                pos_detect = false;
                for j = 1:length(test)
                    frq_fault = 60*freqs(test(j))/r_bins(i);
                    frq_pos = [frq_pos;frq_fault];

                    if (((frq_fault > 0.75) && (frq_fault < 1.25)) ||
((frq_fault > 2.75) && (frq_fault < 3.25))) && ~pos_detect
                        pos_detect = true;
                        if r_bins(i) > 140
                            num_pos = num_pos + 1;
                        end
                    end
                end

                if show_faults
                    figure(1);
                    clf
                    hold on
                    plot(freqs,amp_test,'-k')
                    plot(freqs,thresh(:,i),'-r')
                    fill([freqs;25;-
25],[thresh(:,i);max(thresh(:,i))+0.04;max(thresh(:,i))+0.04],'r','Face
Alpha','0.1','LineStyle','None')
                    pause
                end
            end
            if r_bins(i) > 120
                num_tests = num_tests+1;
            end


        end
    end

    nums = [num_tests num_pos];
end
```

4. FFT_array.m

```
%Creates an FFT of the given data every N datapoints
%Fs = Sample frequency
%Data = Input data vector
%N    = Number Datapoints per FFT
function [freqs,amps] = FFT_array(data,Fs,N)

if ~any(size(data)==1)
    error('ERROR: Input data must be a 1-by-N or N-by-1 vector')
end

data = data - mean(data);

num_fft = floor(length(data)/N);

a = 2;
b = 1;

for i = 1:num_fft
    data_i = data(i*N-(N-1):i*N);

    ffts(:,i) = fft(data_i,N);
    amp_zero = abs((ffts(1,i)).^b/N);
    amp_pos = a.*abs((ffts(2:N/2,i)).^b/N);
    amp_neg = a.*abs((ffts(N/2+1:end,i)).^b/N);

    amps(:,i) = [amp_neg;amp_zero;amp_pos;];

end

freqs_re = Fs/N*linspace(0,N/2,N/2+1);
freqs_ng = Fs/N*linspace(-N/2,0,N/2+1);
freqs = [freqs_ng(1:end-1) freqs_re(1:end-1)]';
```

G. Order Analysis FFT Code

1. Interp_Angle_2.m

```
%/=========================================================================\
%|   PURPOSE: Interpolates accceleration data from constant time-step     |
%|            format to a constant angle-step format                       |
%|-------------------------------------------------------------------------|
%|   INPUTS:          life - Acceleration data from the LifeLine           |
%|                    data - Data output from the squirrel DAQ             |
%|-------------------------------------------------------------------------|
%|   OUTPUTS:            t - Vector of times interpolated to               |
%|            accel_interp - Interpolated acceleration                     |
%|              rot_interp - Interpolated rotor speed                      |
%|-------------------------------------------------------------------------|
%|   Luke Costello, 9/12/20                                                |
%\=========================================================================/
function [t,accel_interp,rot_interp] = Interp_Angle_2(life,data)

%Load Properties
d = ReVIm_prop();
        k = d(2);
accel_col = d(3);
   t_incr = d(5);


accel = life(:,accel_col);

[data_exp,~] = expandData(data,life);
rot_exp = data_exp(:,3);

if length(accel) > length(rot_exp)
    diff = length(accel) - length(rot_exp);
    extras = rot_exp(end-diff+1 : end);
    rot_exp = [rot_exp;extras];

    accel = accel(1:length(rot_exp));
elseif length(rot_exp) > length(accel)
    rot_exp = rot_exp(1:length(accel));
end

%Create list of time vectors to interpolate to
t = [data(1,1)];
i = 1;

while t(end)<life(end,1)
    if i > length(data)
        break
    end

    data_vec = data(i,:);
    time = data_vec(1);
    RPM = data_vec(3);

    omeg = RPM*2*pi/60;
    if omeg < 0
        omeg = -omeg;
```

```
    end

    tht_diff = 2*pi/k;
    t_diff = tht_diff/omeg;

    while t(end) < time + t_incr
        t = [t;t(end) + t_diff];
    end
    i=i+1;
end

%Preallocate Memory
accel_interp = zeros(1,length(t));
rot_interp = zeros(1,length(t));
time_uninterp = life(:,1);

%Loop over times to interpolate to
for i = 1:length(t)-1
    time_interp = t(i);
    %Find the index such that t(index) < t(ik) < t(index+1)
    [~,ind] = min(abs(time_uninterp - time_interp));
    if ind == length(time_uninterp)
        ind = ind-1;
    elseif time_interp < time_uninterp(ind)
        ind = ind-1;
    end

    %Interpolate Accels
    accel_interp(i) = accel(ind) + (t(i) - life(ind,1))/(life(ind+1,1)
- life(ind))*(accel(ind+1) - accel(ind));
    rot_interp(i) = rot_exp(ind) + (t(i) - life(ind,1))/(life(ind+1,1)
- life(ind))*(rot_exp(ind+1) - rot_exp(ind));


    %Panic?
    panic_plot = false;
    if panic_plot
        accel_plt = [accel(ind) accel(ind+1)];
        time_plt = [life(ind,1) life(ind+1,1)];
        figure(10)
        clf
        hold on
        plot(time_plt,accel_plt,'k')
        plot(t(i),accel_interp(i),'ro')
        legend('Uninterpolated data','Interpolated Datapoint')
        pause
    end
end
```

2. FFT_vecs.m

```
%/=========================================================================\
%|   PURPOSE: Calculates the complex vector outputs of an FFT              |
%|-------------------------------------------------------------------------|
%|   INPUTS:    accel - Acceleration vector                                |
%|                  n - Size of FFT to take                                |
%|                rot - Rotor speed vector                                 |
%|-------------------------------------------------------------------------|
%|   OUTPUTS:   freqs - Frequencies of vectors                             |
%|            vectors - Complex FFT vectors                                |
%|               rots - Average rotor speed of each FFT                    |
%|-------------------------------------------------------------------------|
%|   Luke Costello, 9/12/20                                                |
%\=========================================================================/
function [freqs,vectors,rots] = FFT_vecs(accel,n,rot)

n = 2^nextpow2(n);
%Find the number of FFTs to take
num_vec = ceil(length(accel)/n);
vectors = zeros(n,num_vec);
rots = zeros(1,num_vec);

%Loop over number of FFTs to take
for i = 1:num_vec
    %Extract n datapoints, or however many are left
    if i*n > length(accel)
        accel_i = accel(i*n-(n-1):end);
        rot_i = rot(i*n-(n-1):end);
    else
        accel_i = accel(i*n-(n-1):i*n);
        rot_i = rot(i*n-(n-1):i*n);
    end
    vecs     = fft(accel_i,n)';
    vecs_0   = vecs(1);
    vecs_pos = vecs(2:(n/2+1));
    vecs_neg = vecs((n/2+2):end);

    vector_combine = [vecs_neg;vecs_0;vecs_pos];

    vectors(:,i) = [vecs_neg;vecs_0;vecs_pos];
    rots(i) = mean(rot_i);
end

d = ReVIm_prop();
Fs = d(2);
freqs = Fs/n*linspace(-n/2,n/2,n)';
```

3. extract_1P.m

```
%Extracts the 1P component from the FFT
function [freqs,vectors] = extract_1P(freqs,vecs)

[~,rng] = ReVIm_prop();
```

```
%Find indices closest to range
[~,ind1] = min(abs(freqs - rng(1)));
[~,ind2] = min(abs(freqs - rng(2)));
%Extract components
freqs = freqs(ind1:ind2,1);
vectors  = vecs(ind1:ind2,1);
```

4. Learn_Components.m

```
%/=====================================================================\
%|  PURPOSE: Learn the complex vectors of acceleration data in the     |
%|           frequency domain. The end result is an complex vector     |
%|           representing the average complex vector, and a circle     |
%|           surrounding this vector.                                  |
%|                                                                     |
%|           It is expected that the FFT of acceleration data has      |
%|           already been taken constant angle steps.                  |
%|---------------------------------------------------------------------|
%|  INPUTS:   accel_fft - FFT of acceleration data                     |
%|---------------------------------------------------------------------|
%|  OUTPUTS:     center - Complex datapoint representing avg value of  |
%|                        vectors                                      |
%|               radius - Radius of alarm circle; new vectors outside  |
%|                        this circle will raise an alarm.             |
%|---------------------------------------------------------------------|
%|  Code by Luke Costello, 8/28/2020                                   |
%\=====================================================================/

function [center, radius] = Learn_Components(accel_fft,K)

d = ReVIm_prop(K);
K = d(6);

Re = real(accel_fft);
Im = imag(accel_fft);

Re_X0 = mean(Re);
Im_X0 = mean(Im);

Re_std = std(Re);
Im_std = std(Im);
max_std = max([Re_std,Im_std]);

center = Re_X0 + Im_X0*1i;
radius = K*max_std;
```

5. compare_component.m

```
%/=====================================================================\
%|   PURPOSE: Compares the FFT vectors to the complex threshold        |
%|---------------------------------------------------------------------|
%|   INPUTS:  vectors - Vectors to compare to threshold                |
%|             center - Center of complex threshold                    |
%|             radius - Radius of complex threshold                    |
%|             alarms - Number of alarms before test                   |
%|              tests - Number of tests before this test               |
%|---------------------------------------------------------------------|
%|   OUTPUTS:  alarms - Number of alarms after this test               |
%|              tests - Number of tests after this test                |
%|---------------------------------------------------------------------|
%|   Luke Costello, 10/12/20                                           |
%\=====================================================================/
function [alarms,tests] =
compare_component(vectors,center,radius,alarms,tests)

center_re = real(center);
center_im = imag(center);


alarm = false;

%subsequent = 0;

for i = 1:(size(vectors,1)*size(vectors,2))
    vec_re = real(vectors(i));
    vec_im = imag(vectors(i));

    dist = sqrt((vec_re - center_re)^2 + (vec_im - center_im)^2);

    %alarm_now = false;
    if dist > radius
        %alarms = alarms + 1;
        alarm = true;
        %alarm_now = true;
    end
    %tests = tests+1;

end

if alarm
    alarms = alarms+1;
end

tests = tests + 1;
```

H. Cross-Validation Study Code

1. CROSSVALIDATION_DRIVER.m

```
%% The central driver script used for executing the cross-validation
%% study of Luke Costello's M.S. thesis.

clear all

%Prints additional debugging info if debug is set to true
global debug %Print extra info for debugging each loop
global loud  %Be loud!
global show_faults %Plot stuff if a fault occurs
debug = true;
loud = true;

fprintf('<strong>GATHERING DATA</strong>\n')

[list] =
find_files('C:\Users\lukec\OneDrive\Documents\2School\MastersThesis\Cod
e\MATLAB\Modeling\Crossvalidation_Study\Datasets\Healthy');
[fault] =
find_files('C:\Users\lukec\OneDrive\Documents\2School\MastersThesis\Cod
e\MATLAB\Modeling\Crossvalidation_Study\Datasets\Faulty');

fault = fault(1);


%Overcomplicated code, creating a matrix of tests to run
num_memorize = 9;               %Number of datasets to save to memory
in = ones(1,length(list))*2;
list_mem = fullfact(in)-1;      %List all possible binary numbers up to
2^(length(list))
list_mem_i = [];
%Remove any number where (Number of 1's =/= num_memorize)
for i = 1:length(list_mem)
    if sum(list_mem(i,:)) == num_memorize
        list_mem_i=[list_mem_i;list_mem(i,:)];
    end
end
list_mem = list_mem_i;



%Column of acceleration to test
accel_col = 8;

%List of tests to run
tests = [3];

%Loop over each test to run

OFFT_mat_out = cell(1,length(list));
AFFT_mat_out = cell(1,length(list));

NSET_SPRT_mat_out = cell(1,length(list));
for x = 1:length(list)
```

```
    close all
    %% Pick Data
    pick_list = find(list_mem(x,:));
    not_pick = find(~list_mem(x,:));

    [data_mem,data_test,data_fault] = pick_data(list,pick_list,fault);

    NSET_SPRT_mat = [];
    AFFT_mat = [];
    OFFT_mat = [];

    if any(tests==1)
        %% NSET + SPRT
        fprintf('\n<strong>BEGIN NSET+SPRT</strong>\n')
        fprintf('Memorizing data...\n')
        tic
        [mem] = memorize(data_mem);
        fprintf('Done memorizing. (%2.2f seconds elapsed)\n',toc)
        %Estimate the sensor values using MSET
        fprintf('\nEstimating data...\n')
        tic
        est_mem = estimate_sensors(data_mem,mem);
        est_test = estimate_sensors(data_test,mem);
        est_fault = estimate_sensors(data_fault,mem);

        %Calculate residuals
        resid_mem = est_mem(accel_col,:)-data_mem(accel_col,:);
        resid_test = est_test(accel_col,:)-data_test(accel_col,:);
        resid_fault = est_fault(accel_col,:)-data_fault(accel_col,:);

        [~,norml,~] = prop();
        rot_test = data_test(2,:).*norml(2);
        rot_fault = data_fault(2,:).*norml(2);

        std_mem = std(resid_mem);
        std_test = std(resid_test);
        fprintf('Standard deviation of mem: %2.6f || Standard deviation
of test: %2.6f\n',std_mem,std_test)


        if loud
            fprintf('(Sum of residual/Residual Length): [MEMORY]  -
%2.2e\n',sum(resid_mem)/length(resid_mem))
            fprintf('                                   [TEST]  -
%2.2e\n',sum(resid_test)/length(resid_test))
            fprintf('                                  [FAULT]  -
%2.2e\n',sum(resid_fault)/length(resid_fault))
        end

        fprintf('Done estimating. (%2.2f seconds elapsed)\n',toc)

        fprintf('\nComputing SPRT...\n')
        tic
        sig_tr = std(resid_mem);


        S = [sig_tr,m*sig_tr,V,alph,beta];
```

```
        ms = 1:1:6;
        Vs = 1:1:6;
        betas = 0.005:0.005:0.2;
        alphs = 0.005:0.005:0.2;

        S = [sig_tr,m*sig_tr,V,alph,beta];

        %Loop over all possible values of m and V
        for i = 1:length(ms)
            m = ms(i);
            for j = 1:length(Vs)
                %Loop over all possible values of alpha and beta
                V = Vs(j);
                for k = 1:length(alphs)
                    alph = alphs(k);
                    for l = 1:length(betas)
                        beta = betas(l);
                        S = [sig_tr,m*sig_tr,V,alph,beta];


                        [alarms_t,~,~] =
test_data(resid_test,S,rot_test);
                        [alarms_f,~,~] =
test_data(resid_fault,S,rot_fault);

                        TN = alarms_t(2,1); %True Negatives
                        TP = alarms_f(2,2); %True Positives
                        FP = alarms_t(2,2); %False Positives
                        FN = alarms_f(2,1); %False Negatives


                        TP_rate = TP/(TP + FN);
                        FP_rate = FP/(FP + TN);

                        Precision = TP/(TP+FP);
                        Recall = TP_rate;

                        F_score =
2*Precision*Recall/(Precision+Recall);

                        NSET_SPRT_mat = [NSET_SPRT_mat; m V alph beta
TP_rate FP_rate F_score];

                    end
                end


            end

            fprintf('%d ',i);


        end

        %plot(NSET_SPRT_mat(:,6),NSET_SPRT_mat(:,5),'o');
```

```
        NSET_SPRT_mat_out{x} = NSET_SPRT_mat;


        %Compute SPRT
        fprintf('\nDone with SPRT. (%2.2f seconds elapsed)\n',toc)

    end

    %% Real vs Imag FFT
    if any(tests==3)
        fprintf('\n<strong>BEGIN RE V. IMAG FFT</strong>\n')
        tic

        num_FFT = 4096;
        K = 4.5;

        fprintf('Learning Healthy Data...\n')
        vecs1P = [];
        for i = 1:length(list(pick_list))

            ind = pick_list(i);
            load(list(ind));
            %
            %Interpolate acceleration to constant rotor-step rather
than
            %constant timestep


            [t,accel_interp,rot_interp] =
Interp_Angle_2(life,data_raw);
            accel_len = length(accel_interp);
            debug = false;
            if debug
                hold on
                plot(t,accel_interp/16384,'ob')
                plot(life(:,1),life(:,4)/16834,'-*r')
                legend('Interpolated','Raw')
                box on
                xlabel('Time [s]')
                ylabel('Acceleration [g]')
                pause
            end

            %Take the FFT of the interpolated data every N datapoints
            [freqs,vectors,rots] =
FFT_vecs(accel_interp,num_FFT,rot_interp);

            %Extract the 1P component from each column of complex
vectors

            for j = 1:size(vectors,2)
                vectors_j = vectors(:,j);
                [freq1P,vec1P] = extract_1P(freqs,vectors_j);
                vecs1P = [vecs1P;vec1P];
            end

            %figure(21);
```

```
            %hold on
            %plot(vecs1P,'o')
            %pause
        end

        for K = 3.5:0.2:5.5
            alarm_num_h = 0;
            tests_h = 0;
            alarm_num_f = 0;
            tests_f = 0;

            %Loop over healthy data to memorize



            %Learn threshold for 1P component vectors
            fprintf('Learning more :D\n')
            [center,radius] = Learn_Components(vecs1P,K);
            fprintf('Done learning.\nTesting healthy data...\n')


            %Loop over healthy data to test
            for i = 1:length(not_pick)
                %Load dataset
                ind = not_pick(i);
                load(list(ind));
                %Interpolate each
                [t,accel_interp,rot_interp] =
Interp_Angle_2(life,data_raw);
                %Take FFT
                [freqs,vecs,rots] =
FFT_vecs(accel_interp,num_FFT,rot_interp);

                vecs1P_t = [];
                alarms = 0;
                subsequent = 0;
                comparisons = 0;
                %Extract 1P component and compare to threshold
                for j = 1:size(vecs,2)
                    vectors_j = vecs(:,j);
                    rots_j = rots(j);

                    [freq1P,vec1P] = extract_1P(freqs,vectors_j);
                    %vecs1P_t = [vecs1P_t;vec1P];
                    if rots_j > 120
                        [alarm_num_h,tests_h] =
compare_component(vec1P,center,radius,alarm_num_h,tests_h);
                    end



                    %if alarm_num ~= 0
                    %    subsequent = subsequent + 1;
                    %    alarms = alarms + 1;
                    %else
                    %    subsequent = 0;
```

```
                    %end

                    %if subsequent > 3
                    %    fprintf('ALARM! Subsequent alarms:
%d\n',subsequent)
                    %end
                    %comparisons = comparisons + 1;
                end
                %SR = (comparisons - alarms)/comparisons;
                %fprintf('Done with set of data. Statistics: \n%d
Comparisons || %d Alarms || %2.2f
Rotorhealth\n\n',comparisons,alarms,SR)

                %Compare to threshold
                %plot_ReIm(vecs1P_t,center,radius,true,22);


            end
            fprintf('Done testing healthy data.\n')
            fprintf('Testing faulty data...\n')


            %Loop over unhealthy data
            for i = 1:length(fault)
                load(fault(i));
                [t,accel_interp,rot_interp] =
Interp_Angle_2(life,data_raw);

                [freqs,vecs,rots] =
FFT_vecs(accel_interp,num_FFT,rot_interp);

                vecs1P_f = [];
                alarms = 0;
                subsequent = 0;
                for j = 1:size(vecs,2)
                    vectors_j = vecs(:,j);
                    rots_j = rots(j);

                    [freq1P,vec1P] = extract_1P(freqs,vectors_j);

                    if rots_j > 120
                        [alarm_num_f,tests_f] =
compare_component(vec1P,center,radius,alarm_num_f,tests_f);
                    end
                end

            end

            FP = alarm_num_h;
            TN = tests_h - alarm_num_h;


            TP = alarm_num_f;
            FN = tests_f - alarm_num_f;


            TP_rate = TP/(TP+FN);
```

```
            Precision = TP/(TP+FP);
            FP_rate = FP/(FP+TN);

            F_score = 2*(Precision*TP_rate)/(Precision+TP_rate);

            OFFT_mat=[OFFT_mat; FP_rate TP_rate F_score K];

            fprintf('FP Rate: %2.2f | TP Rate: %2.2f | F_score: %2.2f
\n',FP_rate,TP_rate,F_score);


        end

        fprintf('Done with Re v. Imag FFT. (%2.2f seconds
elapsed)\n',toc)
    end

    OFFT_mat_out{x} = OFFT_mat;

    pause
    %AFFT Test
    if any(tests==4)
        fprintf('\n<strong>BEGIN AFFT</strong>\n')
        tic

        for K_thr = 1:0.1:4
            %%Learn healthy data
            num_fft = 1024;

            [~,r_bins] = AFFT_prop();

            threshold = zeros(num_fft,length(r_bins));
            debug = false;
            for i = 1:length(list(pick_list))
                ind=pick_list(i);
                load(list(ind))

                %Expand data to length of lifeline signal
                [data_exp,~] = expand(data_raw,life);

                %Delete extra data from lifeline, and save the lifeline
time column to
                %the data time column
                life = life(1:size(data_exp,1),:);
                data_exp(:,1) = life(:,1);

                [freqs,amps] = FFT_array(life(:,4)/16384,50,num_fft);
                rots_ar = zeros(1,floor(length(data_exp)/num_fft));
                for j = 1:floor(length(data_exp)/num_fft)
                    low = j*num_fft - (num_fft-1);
                    high = j*num_fft;

                    rots = data_exp(low:high,3);
                    rots_ar(j) = mean(rots);
                end
```

```
                [threshold] =
learn_thresh2(amps,freqs,K_thr,rots_ar,threshold);
            end

            debug = false;
            show_faults = false;
            nums_h = [0 0]; %[num_tested,num_positive]
            frq_pos_h = [];
            for i = 1:length(not_pick)
                ind = not_pick(i);
                load(list(ind))

                %Expand data to length of lifeline signal
                [data_exp,~] = expand(data_raw,life);

                %Delete extra data from lifeline, and save the lifeline
time
                column to
                %the data time column
                life = life(1:size(data_exp,1),:);
                data_exp(:,1) = life(:,1);

                [freqs,amps] = FFT_array(life(:,4)/16384,50,num_fft);
                rots_ar = zeros(1,floor(length(data_exp)/num_fft));
                for j = 1:floor(length(data_exp)/num_fft)
                    low = j*num_fft - (num_fft-1);
                    high = j*num_fft;

                    rots = data_exp(low:high,3);
                    rots_ar(j) = mean(rots);
                end

                [nums_h,frq_pos_h] =
test_thresh2(threshold,freqs,amps,rots_ar,r_bins,nums_h,frq_pos_h);
            end

            debug = false;
            show_faults = false;
            nums_f = [0 0]; %[num_tested,num_positive]
            frq_pos_f = [];
            for i = 1:length(fault)
                load(fault(i))

                %Expand data to length of lifeline signal
                [data_exp,~] = expand(data_raw,life);

                %Delete extra data from lifeline, and save the lifeline
time
                column to
                %the data time column
                life = life(1:size(data_exp,1),:);
                data_exp(:,1) = life(:,1);

                [freqs,amps] = FFT_array(life(:,4)/16384,50,num_fft);
                rots_ar = zeros(1,floor(length(data_exp)/num_fft));
                for j = 1:floor(length(data_exp)/num_fft)
                    low = j*num_fft - (num_fft-1);
```

```
                    high = j*num_fft;

                    rots = data_exp(low:high,3);
                    rots_ar(j) = mean(rots);
                end

                [nums_f,frq_pos_f] =
test_thresh2(threshold,freqs,amps,rots_ar,r_bins,nums_f,frq_pos_f);
            end

            TP = nums_f(2);             %True positives
            FN = nums_f(1) - nums_f(2); %False negatives
            TN = nums_h(1) - nums_h(2); %True negatives
            FP = nums_h(2);             %False positives


            TP_rate = TP/(TP+FN);
            Precision = TP/(TP+FP);
            FP_rate = FP/(FP+TN);

            F_score = 2*(Precision*TP_rate)/(Precision+TP_rate);

            AFFT_mat=[AFFT_mat; FP_rate TP_rate K_thr F_score];

        end
        plot(AFFT_mat(:,1),AFFT_mat(:,2),'o');
        fprintf('Done with AFFT. (%2.2f seconds elapsed)\n',toc)
    end
    AFFT_mat_out{x} = AFFT_mat;

    %LSCh Test
    if any(tests==5)
        slopes = [];
        intercepts = [];
        figure(2)
        hold on
        RMS = [];
        pow = [];
        fprintf('Calc-ing a healthy\n')
        for i = 1:length(list)
            load(list(i))
            data(end-10:end,:) = [];
            RMS = [RMS;data(:,10)];
            pow = [pow;data(:,4).*data(:,5)];
        end
        RMS = RMS./16384;


        RMS_av = [];
        pow_av = [];
        for i = 1:ceil(length(RMS)/60)
            low = i*60-59;
            high = i*60;
            if high>length(RMS)
                high = length(RMS);
            end
```

```
    RMSs = RMS(low:high,:);
    RMS_av = [RMS_av mean(RMSs)];

    pows = pow(low:high,:);
    pow_av = [pow_av mean(pows)];
end


[slope,intercept] = GetParams(RMS_av,pow_av);
slopes = [slopes slope];
intercepts = [intercepts intercept];
Xs = 0:0.001:1;
Ys = slope*Xs + intercept;
plot(RMS_av,pow_av,'b.',Xs,Ys,'b-');
pause


RMS = [];
pow = [];
fprintf('Calc-ing a fault\n')
for i = 1:length(fault)
    load(fault(i))
    data(end-10:end,:) = [];
    RMS = [RMS;data(:,10)];
    pow = [pow;data(:,4).*data(:,5)];
end
RMS = RMS./16384;

RMS_av = [];
pow_av = [];
for i = 1:ceil(length(RMS)/60)
    low = i*60-59;
    high = i*60;
    if high>length(RMS)
        high = length(RMS);
    end

    RMSs = RMS(low:high,:);
    RMS_av = [RMS_av mean(RMSs)];

    pows = pow(low:high,:);
    pow_av = [pow_av mean(pows)];
end

[slope,intercept] = GetParams(RMS_av,pow_av);
slopes = [slopes slope];
intercepts = [intercepts intercept];
Xs = 0:0.001:1;
Ys = slope*Xs + intercept;
plot(RMS_av,pow_av,'r.',Xs,Ys,'r-');

box on
xlim([0 0.2])
ylim([0 1000])
xlabel('RMS Acceleration [g]')
ylabel('Power Output [W]')
```

```
        legend('Healthy Data','Healthy Regression','Imbalanced
Data','Imbalanced Regression','Location','EastOutside')
        pause

        figure(1);
        subplot(2,1,1)
        plot(slopes)
        subplot(2,1,2)
        plot(intercepts)
    end

end

%{
load('NSET_out.mat')
load('Output.mat')

figure(1)
clf
hold on
plot(NSET_SPRT_mat(:,6),NSET_SPRT_mat(:,5),'^g')
plot(AFFT_mat(:,1),AFFT_mat(:,2),'ob')
plot(OFFT_mat(:,1),OFFT_mat(:,2),'*r')
legend('NSET+SPRT','AFFT','OFFT')
box on
xlabel('False Positive Rate')
ylabel('True Positive Rate')
%}
out = zeros([size(NSET_SPRT_mat),10]);

for i = 1:10
    out(:,:,i) = cell2mat(NSET_SPRT_mat_out(i));
end

out_avg = mean(out,3);
plot(out_avg(:,6),out_avg(:,5),'.')
```

I. Python Implementation Code

NSET Code

```
import time
import math

def load(fileName):
    '''Load a text file into memory.'''
    out = []
    with open(fileName) as txt:
        for line in txt:
            line_str = line.split(',')
            line_num = [float(i) for i in line_str]
            out.append(line_num)

    return out

def RMS(vec):
    '''Computes the RMS value of signal'''
    if type(vec[0])!=float and type(vec[0])!=int:
        raise TypeError('Error: Input is not a list containing numbers')

    sums = 0
    for x in vec:
        sqr = x**2
        sums+=sqr

    RMS_out = math.sqrt(sums/len(vec))
    return RMS_out

def LL(vec,t_len=0.02):
    '''Computes the line length of a signal'''
    n = len(vec)
    if type(vec[0])!=float and type(vec[0])!=int:
        raise TypeError('Error: Input is not a list containing numbers')
    dX = t_len
    dX_2 = dX**2
    sums = 0
    for i in range(1,len(vec)):

        dY = vec[i] - vec[i-1]
        dist = math.sqrt(dX_2 + dY**2)
        sums+=dist
    return sums
```

```
def bun(A,B):
    '''Computes the nonlinear operator of two vectors, A and B.
    Currently, the nonlinear operator is simply the euclidean distance
    between the two vectors-- |A-B|'''
    if len(A) != len(B):
        raise IndexError('Error: Matrices are not identical in length!')

    sum_full = 0
    for i in range(0,len(A)):
        sum_i = (A[i] - B[i])**2
        sum_full += sum_i

    return (sum_full)**0.5

def bun_mat(A,B):
    '''Computes the nonlinear operator of two matrices, A and B.
    Note that if A or B is simply a vector/list, it must be made
    into a list of lists of length 1, ie [[1,2,3]]'''
    out_mat = []
    for i in range(0,len(A)):
        A_vec = A[i]
        row_i = []
        for j in range(0,len(B)):
            B_vec = B[j]

            bun_i_j = bun(A_vec,B_vec)
            row_i.append(bun_i_j)

        if len(row_i)==1:
            row_i=row_i[0]
        out_mat.append(row_i)

    return out_mat

def mult_mat_vec(mat,vec):
    '''Compute the operation OUT = Mat*Vec, where OUT is the output,
    Mat is an m-by-n matrix, and vec is an n-by-1 vector.'''
    if len(mat[0]) != len(vec):
        raise IndexError('Error: Matrix and vector cannot be multiplied - Dime
nsions do not match')

    out = []
    for i in range(0,len(mat)):
```

```
        mat_row = mat[i]
        sum = 0
        for j in range(0,len(vec)):
            sum+=mat_row[j]*vec[j]
        out.append(sum)

    return out


def transp(mat):
    '''Compute the transpose of an m-by-n matrix.'''
    i_len = len(mat)
    j_len = len(mat[0])

    #Form output matrix
    out = []
    for j in range(0,j_len):
        out.append([])
        for i in range(0,i_len):
            out[j].append(0)

    for i in range(0,i_len):
        for j in range(0,j_len):
            out[j][i]=mat[i][j]

    return out

def estimate(D,inv,X_obs):

    rhs = bun_mat(transp(D),[X_obs])
    wght = mult_mat_vec(inv,rhs)
    X_est = mult_mat_vec(D,wght)

    return X_est

class NSET:
    '''Creates an NSET object to simplify calculating the residual.
    The __init__ function requires 3 arguments: D, invDbunDm and est_num.
        D:        The filename for the memory matrix. Must be a comma-
delimited file;
                  designed for a .txt and will probably work with a .csv. A st
ring is expected.

        invDbunD: The filename for the operation inverse[D_transpose (bun) D].
                  Same input requirements as D.
```

```
        est_num:  The row of the memory matrix and observed state vector to es
timate.

        nrml:     The array to normalize observed data vectors by (NOTE: This
MUST be identical to
                  the training model.)'''


    def __init__(self,D,invDbunD,est_num,nrml):
        self.D = load(D)
        self.invDbunD = load(invDbunD)
        self.est_num = est_num
        self.nrml = nrml

        self.fault_flag = False
        self.has_tested = False
        self.last_five = [0,0,0,0,0]

    def calc_resid(self,X_obs_big):
        X_obs = [X/n for X,n in zip(X_obs_big,self.nrml)]
        X_est = estimate(self.D,self.invDbunD,X_obs)
        resid = X_est[self.est_num] - X_obs[self.est_num]
        return resid
```

SPRT Code

```
import math

##Set SPRT parameters
alf = 0.01   #False alarm rate
bet = 0.005  #Missed alarm rate
m = 3
V = 2

std_mem = 0.012760418 #Std Dev of Residual when testing mem
M = m*std_mem

class SPRT:
    def __init__(self,alf=0.01,bet=0.005,m=3,V=2,std_mem=0.012760418):
        '''Initialize the properties of the SPRT test'''
        self.alf = alf
        self.bet = bet
        self.std_mem = std_mem

        self.A = math.log(bet/(1-alf))
        self.B = math.log((1-bet)/alf)
        self.V   = V
        self.M   = m*std_mem

        self.num_healthy = 0
        self.num_faulty  = 0

        self.SPRT_ind = [0,0]




    def calc_index(self,resid):
        this_Fault = False

        if (self.SPRT_ind[0] == self.A) or (self.SPRT_ind[0] == self.B):
            self.SPRT_ind[0] == 0

        if (self.SPRT_ind[1] == self.A) or (self.SPRT_ind[1] == self.B):
            self.SPRT_ind[1] == 0

        self.SPRT_ind[0] = self.SPRT_ind[0] + (self.M / (self.std_mem)**2)*( r
esid - self.M/2)
```

```
        self.SPRT_ind[1] = self.SPRT_ind[1] + (self.M / (self.std_mem)**2)*(-
resid - self.M/2)

        if self.SPRT_ind[0] >= self.B:
            self.SPRT_ind[0] = self.B
            self.num_healthy += 1
        elif self.SPRT_ind[0] <= self.A:
            self.SPRT_ind[0] = self.A

        if self.SPRT_ind[1] >= self.B:
            self.SPRT_ind[1] = self.B
            self.num_faulty += 1
            this_Fault = True
        elif self.SPRT_ind[1] <= self.A:
            self.SPRT_ind[1] = self.A

        return this_Fault

    def reset_index(self):
        self.SPRT_ind = [0,0]
```

AFFT Code

```
import numpy as np
import numpy.fft as fft
import math


class AFFT:
    '''This class compares the FFT of nacelle vibrations with an adaptive
    threshold based on rotorspeed. For each increment in rotorspeed, a
    unique adaptive threshold is selected and compared against the FFT
    output. The thresholds are saved in the file "AFFT_threshold.txt" and
    may be generated using the MATLAB program "GenerateTextFiles.mlapp"
    within the FDC_FileGenerationTools folder in Luke Costello's
    MS Thesis directory.
    For practical usage, one must first initiate an AFFT object using this
    class and relevant parameters, then load a text file using the
    command AFFT.load(filename). Finally, AFFT testing may be done using
    the command AFFT.examine(), which returns a list of frequencies
    (at multiples of the rotorspeed) that faults are detected.
    Care must be taken such that:
        1. The parameters in the __init__ function are identical to those
           used to create the thresholds
        2. The accelerations variable supplied to examine is identical to
           in FFT length as those used to train the model
    You can refer to a specific location in a threshold using
    AFFT.thr[i][j], where i is the threshold number, and j is the
    location in the threshold.'''
    def __init__(self,Fs=50,RPM_lo=32.5,RPM_hi=212.5,RPM_step=5,XP_monitor
 = [1,3],XP_bin=0.5):
        self.thr = []
        self.thr_short = []

        self.Fs=Fs
        self.RPMs = np.arange(RPM_lo,RPM_hi,RPM_step)

        self.threshs = []
        self.lastFreq = []
        self.lastSpect = []
        self.lastFreq_short = []
        self.lastSpect_short = []

        self.XP_monitor = XP_monitor
        self.XP_bin = XP_bin
```

```
        self.fault_flag = False
        self.has_tested = False

        self.last_five = [0,0,0,0,0]

    def load(self,fileName):
        '''Loads the fileName file into object memory.'''
        self.threshs = []
        self.RPMs = []
        with open(fileName) as file:
            for line in file:
                line = line[0:-1]
                if line[0]=='#':
                    line_list = line.split(',')

                    nums = [float(x) for x in line_list[1:]]

                    self.RPMs.append(float(line_list[0][1:]))
                    self.threshs.append(nums)



    def selectThreshold(self,av_RPM):
        '''Selects the adaptive threshold closest to the average
           rotorspeed input as av_RPM'''
        compared = np.abs([x-av_RPM for x in self.RPMs])
        ind = compared.argmin()
        self.thr = self.threshs[ind]

    def compareToThreshold(self,spect):
        '''Compares the FFT output to the adaptive threshold selected.'''
        if self.thr_short == []:
            raise ValueError('Threshold not yet defined!')

        log = [ (spect[i] - self.thr_short[i])>=0 for i in range(0,len(spe
ct))]
        fault = [i for i, n in enumerate(log) if n==True]


        return fault


    def FFT(self,sig):

        N = len(sig)
```

```
        sig_mean = sum(sig)/len(sig)
        sig_corr = [x - sig_mean for x in sig]
        spect = np.abs((fft.fft(sig_corr,N)))/N
        freq = fft.fftfreq(len(sig_corr),d=1/self.Fs)

        return [freq,spect]

    def extractXP(self,RPM_av,freq,spect):
        '''Extracts the frequency components in terms of multiples of the
rotorspeed as specified by the XP_monitor variable.'''
        RPM_freq = RPM_av/60
        self.thr_short = []
        freq_short = []
        spect_short = []

        for XP in self.XP_monitor:
            freq_lo = RPM_freq*XP - self.XP_bin
            freq_hi = RPM_freq*XP + self.XP_bin

            compare_lo = np.abs([x-freq_lo for x in freq])
            ind_lo = np.argmin(compare_lo)
            compare_hi = np.abs([x-freq_hi for x in freq])
            ind_hi = np.argmin(compare_hi)

            [self.thr_short.append(x) for x in self.thr[ind_lo:ind_hi]]
            [freq_short.append(x) for x in freq[ind_lo:ind_hi]]
            [spect_short.append(x) for x in spect[ind_lo:ind_hi]]

        return [freq_short,spect_short]

    def examine(self,accels,RPMs):
        '''Use this command for using the AFFT algorithm in practical use.
        Selects the adaptive threshold for use, computes the FFT of the
        input data, extracts the desired frequency components, then
        compares the FFT output to the selected threshold.
        Returns any faults as multiples of the rotorspeed.'''
        #Compute average rotorspeed
        RPM_av = sum(RPMs)/len(RPMs)
        #Select threshold
        self.selectThreshold(RPM_av)
        #Compute FFT
        [freq,spect] = self.FFT(accels)
        self.lastFreq = freq
        self.lastSpect = spect
        #Extract desired frequencies, and compare to threshold
```

```
[freq_short,spect_short] = self.extractXP(RPM_av,freq,spect)
self.lastFreq_short = freq_short
self.lastSpect_short = spect_short
fault = self.compareToThreshold(spect_short)
#Convert the faulty frequencies to multiples of rotorspeed
fault_freqs = [freq_short[i]*60/RPM_av for i in fault]

return [fault_freqs]
```

Example Usage

```
import NSET as N
import SPRT as S
import AFFT as A
import math
from matplotlib import pyplot as plt

import numpy as np

'''The decision-making in this file is nearly identical to that of
the lifeline_FDC.py file and exists to help the user test the
implementation of the Lifeline CMS algorithms: NSET+SPRT & AFFT.'''

log_freq = 50

nrml = [300,300*5,4*1024,4*1024,18*1024,5E5,5E5,5E5]
NSET_obj = N.NSET('NSET_memory.txt','NSET_inverse.txt',6,nrml)
SPRT_obj = S.SPRT()
AFFT_obj = A.AFFT()
AFFT_obj.load('AFFT_threshold.txt')

def run_crossvalidation(name,params):

    alf = params[0]
    bet = params[1]
    m = params[2]
    V = params[3]

    K_thr = params[4]

    SPRT_obj.alf = alf
    SPRT_obj.bet = bet
    SPRT_obj.M = m*SPRT_obj.std_mem
    SPRT_obj.V = V

    SPRT_obj.reset_index()


    tests = 0
    alarms = 0

    index_val = []
    xaccels = []
    yaccels = []
```

```
    zaccels = []
    rot_speeds = []
    volts = []
    amps = []
    SPRT_indices = []


    with open(name) as data:
        for line in data:

            if line[0] == '~':
                values = line.split(',')

                if len(values) == 7:

                    try:
                        index_val.append(values[0])
                        xaccels.append((float)(values[1]))
                        yaccels.append((float)(values[2]))
                        zaccels.append((float)(values[3]))

                        rot_speeds.append((float)(values[4]))

                        volts.append((float)(values[5]))
                        amps.append((float)(values[6]))
                    except:
                        print('FIRE!!!!')

                    if (len(yaccels)%500 == 0):
                        xaccels_short = xaccels[-500:]
                        yaccels_short = yaccels[-500:]
                        zaccels_short = zaccels[-500:]
                        rot_speeds_short = rot_speeds[-500:]
                        volts_short = volts[-500:]
                        amps_short = amps[-500:]

                        #Calculate the RMS value
                        x_RMS = math.sqrt( sum([x**2 for x in xaccels_shor
t] )/500 )
                        y_RMS = math.sqrt( sum([y**2 for y in yaccels_shor
t] )/500 )
                        z_RMS = math.sqrt( sum([z**2 for z in zaccels_shor
t] )/500 )

                        #Calculate Line Length values
```

```
                        x_LL = sum( [ math.sqrt((x_i - x_j)**2 + (1/log_fr
eq)**2) for x_i,x_j in zip(xaccels_short[1:],xaccels_short[0:-1])])
                        y_LL = sum( [ math.sqrt((y_i - y_j)**2 + (1/log_fr
eq)**2) for y_i,y_j in zip(yaccels_short[1:],yaccels_short[0:-1])])
                        z_LL = sum( [ math.sqrt((z_i - z_j)**2 + (1/log_fr
eq)**2) for z_i,z_j in zip(zaccels_short[1:],zaccels_short[0:-1])])

                        pows = [x*y for x,y in zip(volts_short,amps_short)
]

                        rot_av = sum(rot_speeds_short)/len(rot_speeds_shor
t)

                        pow_av = sum(pows)/len(pows)

                        X_obs_lrg = [rot_av,pow_av,x_RMS,y_RMS,z_RMS,x_LL,
y_LL,z_LL]

                        if rot_av >= 120:
                            resid = NSET_obj.calc_resid(X_obs_lrg)
                            fault = SPRT_obj.calc_index(resid)

                            if fault:
                                NSET_obj.fault_flag = True
                                NSET_obj.last_five.append(1)
                            else:
                                NSET_obj.fault_flag = False
                                NSET_obj.last_five.append(0)

                            NSET_obj.last_five.pop(0)

                            SPRT_indices.append(SPRT_obj.SPRT_ind[1])


                        NSET_obj.has_tested = True

                    if (len(yaccels)%1024 == 0):
                        yaccels_long = yaccels[-1024:]
                        rot_speeds_long = rot_speeds[-1024:]

                        rot_av = sum(rot_speeds_long)/len(rot_speeds_long)

                        if rot_av >= 120:
```

```
                            [fault_freqs] = AFFT_obj.examine(yaccels_long,
rot_speeds_long)

                            if fault_freqs:
                                AFFT_obj.fault_flag = True
                                AFFT_obj.last_five.append(1)
                            else:
                                AFFT_obj.fault_flag = False
                                AFFT_obj.last_five.append(0)

                            AFFT_obj.last_five.pop(0)


                            AFFT_obj.has_tested = True

                            #plt.plot(AFFT_obj.lastFreq,AFFT_obj.lastSpect
,AFFT_obj.lastFreq,AFFT_obj.thr)
                            #plt.show()

                    if NSET_obj.has_tested and AFFT_obj.has_tested:

                        if NSET_obj.fault_flag and AFFT_obj.fault_flag:
                            alarms += 1

                        elif sum (NSET_obj.last_five) > 4:
                            alarms += 1

                        elif sum (AFFT_obj.last_five) > 4:
                            alarms += 1

                        NSET_obj.has_tested = False
                        AFFT_obj.has_tested = False

                        tests += 1

    #plt.plot(SPRT_indices)
    #plt.show()

    return [tests,alarms]


alfs = np.arange(0.0025,0.0225,0.0025)
bets = np.arange(0.0025,0.0225,0.0025)
ms = np.arange(0.25,2.25,0.25)
Vs = np.arange(2,3,1)
```

```
K_thrs = [1]



alfs = [0.005]
bets = [0.01]
ms = [2]
Vs = [1]
K_thrs = [1]



tuning_array = []#['TP_rate,FP_rate,Accuracy,F_score']
FP_rates = []

TP_rates = []

healthys = ['H1.txt','H2.txt','H3.txt','H4.txt','H5.txt','H6.txt','H7.txt'
,'H8.txt','H9.txt','H10.txt']


for alf in alfs:
    for bet in bets:
        for m in ms:
            for V in Vs:
                for K_thr in K_thrs:

                    single_array = []

                    for healthy in healthys:

                        params = [alf,bet,m,V,K_thr]

                        faulty = 'Unbalanced.txt'

                        [tests_healthy,alarms_healthy] = run_crossvalidati
on(healthy,params)

                        [tests_faulty,alarms_faulty] = run_crossvalidation
(faulty,params)

                        TN = tests_healthy - alarms_healthy
                        TP = alarms_faulty
                        FN = tests_faulty - alarms_faulty
                        FP = alarms_healthy
```

```
                        TP_rate = TP/(TP+FN)
                        FP_rate = FP/(FP+TN)

                        accuracy = (TP+TN)/(TP+TN+FP+FN)

                        #print(str(TP_rate)+' '+str(FP_rate))
                        #print(accuracy)

                        Precision = TP/(TP+FP)
                        F_score = 2*Precision*TP_rate/(Precision+TP_rate)

                        single_array.append([TP_rate,FP_rate,accuracy,F_sc
ore,alf,bet,m,V])

                    sums = [0,0,0,0,0,0,0,0]

                    for row in single_array:
                        ind = 0
                        for num in row:
                            sums[ind] += num
                            ind += 1

                    avg = [x/len(single_array) for x in sums]
                    tuning_array.append(avg)

                    print( 'TP Rate: ' + str(tuning_array[-
1][0]) + ' | FP Rate:' + str(tuning_array[-1][1]) )
                    FP_rates.append(tuning_array[-1][1])
                    TP_rates.append(tuning_array[-1][0])

#plt.plot(FP_rates,TP_rates)
#plt.show()

with open('output.txt','w') as file:
    file.write('TP_rate,FP_rate,Accuracy,F_score,Alpha,Beta,m,V\n')
    for row in tuning_array:
        row_rounded = [round(x,3) for x in row]
        file.write( ','.join(map(str,row_rounded)) + '\n' )
```