DEVELOPMENT OF A MODEL AND IMBALANCE DETECTION SYSTEM FOR THE CAL POLY WIND TURBINE

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

Ryan Takatsuka

June 2019

COMMITTEE MEMBERSHIP

TITLE:	Development of a Model and Imbalance Detection System for the Cal Poly Wind Turbine
AUTHOR:	Ryan Takatsuka
DATE SUBMITTED:	June 2019
COMMITTEE CHAIR:	John Ridgely, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Glen Thorncroft, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Patrick Lemieux, Ph.D. Professor of Mechanical Engineering

ABSTRACT

Development of a Model and Imbalance Detection System for the Cal Poly Wind Turbine

Ryan Takatsuka

This thesis develops a model of the Cal Poly Wind Turbine that is used to determine if there is an imbalance in the turbine rotor. A theoretical model is derived to estimate the expected vibrations when there is an imbalance in the rotor. Vibration and acceleration data are collected from the turbine tower during operation to confirm the model is useful and accurate for determining imbalances in the turbine.

Digital signal processing techniques for analyzing the vibration data are explored and tested with simulation data. This includes frequency shifts, lock-in amplifiers, phase-locked loops, discrete Fourier transforms, and decimation filters. The processed data is fed into an algorithm that determines if there is an imbalance.

The detection algorithm consists of a machine learning classification model that uses experimental data to train and increase the success rate of the imbalance detection. Various models are explored, including the K-Nearest Neighbors algorithm, logistic regression, and neural networks. These models have trade-offs between mathematical complexity, required computing power, scalability, and accuracy. With proper implementations of these detection models, the imbalance detection accuracy was measured to be about 90%.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 System Overview

The wind turbine that is analyzed in this thesis is part of the Cal Poly Wind Power Research Center. This is designed for research into smaller wind turbines and to educate engineers in all aspects of the wind power industry. The Cal Poly Wind Turbine Tower supports a 3 kW Horizontal-Axis Wind Turbine, and the tower was analyzed by Tae-gyun (Tom) Gwon[17]. In Gwon's thesis, an ABAQUS model of the tower is developed to analyze natural frequencies and vibrations. A finite-element model, such as this one, is too complicated to run on a cheap microcontroller; however, the results of this model can be compared with the simple lumped-parameter model to determine the validity of the simplifications.

The Cal Poly Wind Turbine has no current method of detecting an imbalance. It is possible to install an intrusive and expensive device that monitors many tower parameters (such as multiple acceleration or strain measurements at various positions on the tower and blades), but it would be much more desirable to have an inexpensive device that can accurately detect an imbalance with minimal setup/installation effort.

1.2 Objective

The primary objective of this thesis is to develop a method for identifying a blade imbalance in the field. This method must be simple enough to perform on a microcontroller in real time, while maintaining the ability to be mounted to any small

scale wind turbine. This will help to prevent any catastrophic failures resulting in damaged blades and an inoperable wind turbine. Additionally, maintenance costs can be reduced because the turbines in good condition will not have to be inspected as often. The main focus of the paper will be introducing a simplified turbine tower model and providing a digital signal processing method for analyzing the data.

A simplified tower model will help to develop and test various signal processing methods. It is difficult and time consuming to obtain experimental data from the tower, so having a tower model will speed up the algorithm design process. Once the signal processing method has been tested and refined on the analytic model, it can then be tested on experimental tower data with minimal tuning.

1.3 Background

Cal Poly's wind turbine is in the Escuela Ranch in an unpopulated area. The tower is a tapered tubular pole made of ASTM A572 Grade-50 Steel and has a tilting feature which allows relatively easy access to the nacelle. The tower is rotated about 2 journal bearings at the base via a winch attached to the Cal Poly Wind Power Research Center (CPWPRC) truck. More details about the tower design and analysis can be found in *Structural Analyses of Wind Turbine Tower for 3 KW Horizontal-Axis Wind Turbine* [17].

1.4 Turbine Failures

The goal of this project is to develop a method for detecting a rotor imbalance that could be potentially harmful to the turbine. This would allow the turbine to be shut down before any failures occur.

SCADA System	WT Make	Technology	Rated Capacity (kW)	Nb of Turbines	Failures per Turbine	Alarms per Turbine
1	A	Geared	1500	55	0.709	4170.07
2	B, C	Dir. Drive	2000	57	0.632	1120.35
3	D	Geared	850	77	2.208	2778.78
4	E	Geared	2000	168	1.780	4704.57
5	F, G	Geared	1800 & 2000	83	1.313	572.14

Figure 1.1: Data used for the SCADA Alarms and Failure Analysis [22]

A common method for tracking and preventing turbine failures is the use of Supervisory Control and Data Acquisition (SCADA) alarms [12]. SCADA is a control system architecture that uses high-level user interfaces networked with peripheral devices (such as PLCs and PID controllers). This is an effective method for predicting turbine failure, but can be expensive and complicated to implement. An example of SCADA alarm results are shown in Figure 1.1. In this table, the turbines are categorized by type and the failure rate is listed along with the number of SCADA alarms. The SCADA systems for this analysis use threshold detection on various parameters of the turbine. About 80% of the failures in this table are caused by turbine components, and only about 2 - 3% of failures are caused by environmental conditions (the remaining failures fall into the "other" category [22]).

According to a study on wind turbine accidents [5], there are 4 main categories of accidents. These include transportation, construction, operation, and maintenance accidents, which can be caused by either nature, human error, or equipment failure. According to the study, operation accidents are significantly more common than any other category of failures [22]. Most of the operation accidents (about 80%) are caused by either equipment failures or nature events. The detection method outlined in this document would hopefully be able to prevent many of the operation-based accidents, which are a majority of the turbine accidents.

1.5 Tower Vibration Research

Wind turbine tower vibrations are important to the health of turbines and have been studied before. In one study, a nonlinear state estimation technique [28] (NSET) is used to attempt to predict wind turbine failures. This model uses SCADA data to relate different operating parameters to the health of the turbine. Typically, many variables are used as inputs to determine turbine health and status because vibration alone isn't always sufficient [28]. When many variables are used, statistical model become significantly more complex and require large-scale optimization processes which is where machine learning models accel.

Most tower vibration analyses are empirically derived because of the stochastic nature of the wind speed and the amount of variables affecting the turbine. Data-driven models [34] are very common in the wind power space, especially with monitoring systems such as SCADA.

1.6 Small turbine monitoring system

As previously discussed, there are a lot of options for large-scale turbine monitoring, but they are expensive and difficult to implement and maintain because many of them need to be directly integrated with the turbine computer to collect the necessary data. Small-scale turbine systems don't have all of the same requirements as the large-scale systems. It is valuable to have a lightweight and inexpensive device that can monitor the health of a turbine without requiring direct integration to internal operation parameters. Ideally, this device should be able to detect impending problems, halt operation, and notify the maintenance operator.

A small-turbine monitoring system would reduce the frequency of maintenance be-

cause it would only need to be checked when the system reports possible trouble. It would also reduce the operation and maintenance cost by stopping all operations when there is a possible problem. Because this device won't have access to internal operating parameters (such as generator power, internal temperature, rotor speed, wind speed, and wind direction), it will need to monitor the turbine vibrations, which can efficiently be done using accelerometers. One common turbine problem is a rotor imbalance, caused by either blade damage, loose fasteners, or build-up material on the blades. If the monitoring device can detect these imbalances, the turbine can be fixed before any additional damage is done.

Additionally, an ideal monitoring device would be wireless and battery-powered. This means the processor should be very low-power and will need very efficient algorithms and data processing methods when acquiring data and making predictions. This document will discuss various signal processing methods and their efficiencies, along with analyzing a few machine learning models for rotor imbalance predictions.

Chapter 2

MATHEMATICAL MODEL

2.1 Assumptions/Problem Statement

The turbine tower is modeled as a cantilever beam with a concentrated mass at the end. The tower is the beam itself, and the nacelle is a concentrated mass at the end of the simplified beam. The tower mass is condensed into an effective lump sum mass and added to the nacelle mass. This allows the tower to be treated as a mass-less cantilever beam.

This problem is treated as forced vibration, where the forcing function is a periodic force caused from an eccentric mass on the rotor. This eccentric mass represents an imbalance in the turbine, and will cause vibrations in the tower. Figure 2.1 shows the simplified wind turbine model with the applied forces. The cross-section of the tower is assumed to taper linearly from the base to the top.

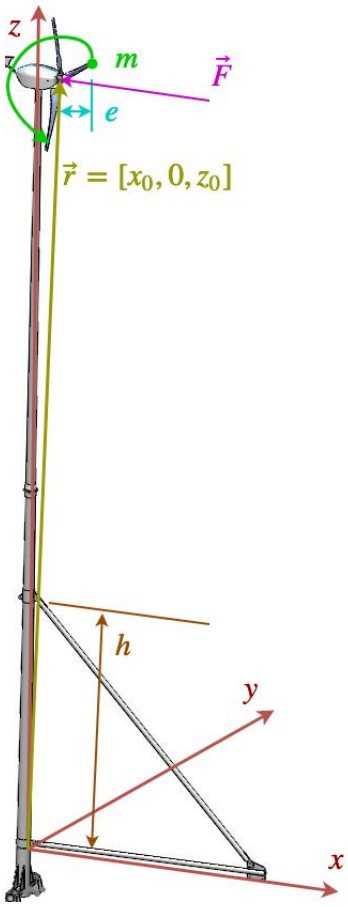

Figure 2.1: The simplified wind turbine model showing the forces from an eccentric mass on the rotor. This figure also shows the coordinate system that will be used for the tower analysis and is created using a Draw.io overlay on a SolidWorks model.

Table 2.1: Variable Definitions

Variable	Description
$\vec{F}_{original}$	The original force applied to the end of the rotor due to an imbalance in the blades
F_0	The magnitude of the force caused by the eccentric mass
ω	The rotational speed of the blades
t	Time
m	Eccentric mass
e	Distance the eccentric mass is from the axis of rotation of the rotor
$\vec{M}$	Moment that the eccentric force applies after the force is translated to the tower axis
$\vec{r}$	The position vector for the top of the rotor with respect to the bottom of the turbine tower ($r = [x_0, 0, z_0]$)
M_x	The moment function in the x-direction
M_y	The moment function in the y-direction
I_b	The area moment of inertia at the base of the tower
I_t	The area moment of inertia at the top of the tower
h	The distance from the base of the tower to the ginpole pin-joint on the tower
E	Modulus of elasticity of the tower (steel)
d_b	Diameter at the base of the tower
d_t	Diameter at the top of the tower

2.2 Calculating Tower Vibrations

2.2.1 Determining Effective Force

An eccentric mass, m, on the rotor causes a force defined by the following equation:

$$\vec{F}(t) = \begin{bmatrix} 0 \\ F_0 \sin(\omega t) \\ F_0 \cos(\omega t) \end{bmatrix} \tag{2.1}$$

where

$$F_0 = me\omega^2 \tag{2.2}$$

In order to treat the system as a cantilever beam, the force has to be acting in the center of the nacelle and on the same axis as the tower. When the force vector (Equation 2.1) is translated to the axis of the tower, a moment needs to be introduced to account for the force vector translation:

$$\vec{M} = \vec{r} \times \vec{F} = \begin{bmatrix} -F_0 z_0 \sin(\omega t) \\ -F_0 x_0 \cos(\omega t) \\ F_0 x_0 \sin(\omega t) \end{bmatrix} \tag{2.3}$$

The moment in (Equation 2.3) creates an effective force at the top of the tower that needs to be added to the original force (Equation 2.1). The moment about the z-axis is ignored because torsional effects on the tower are negligible when determining nacelle displacement. This results in the following effective force on the top of the tower:

$$\vec{F}_{effective} = \vec{F}_{original} + \vec{F}_{moment} \tag{2.4}$$

$$\vec{F}_{effective} = \begin{bmatrix} 0 \\ F_0 \sin(\omega t) \\ F_0 \cos(\omega t) \end{bmatrix} + \begin{bmatrix} -\frac{F_0 x_0}{z_0} \cos(\omega t) \\ F_0 \sin(\omega t) \\ 0 \end{bmatrix}$$

$$\vec{F}_{effective} = \begin{bmatrix} -\frac{F_0 x_0}{z_0} \cos(\omega t) \\ 2F_0 \sin(\omega t) \\ F_0 \cos(\omega t) \end{bmatrix} \tag{2.5}$$

2.2.2 Determining Effective Spring Constant

The tower model has a different spring constant for the x and y directions. In the y direction, the gin pole has a negligible effect on the stiffness of the tower, so the tower acts as a tapered cantilever beam.

2.2.2.1 Mathematical derivation of the y-direction spring constant, K_y

Assuming the tower acts like a cantilever beam in this direction with a concentrated force at the end of the beam, the moment, $M_x(z)$ can be expressed as:

$$M_x(z) = F_y\, z - F_y\, z_0 \tag{2.6}$$

Because the beam is linearly tapered, the diameter, d, can be written as a function of tower height, z_0.

$$d(z) = d_b - \frac{z\ (d_b - d_t)}{z_0} \tag{2.7}$$

d_b is the diameter of the tower at the bottom, and d_t is the diameter of the tower at the top. The moment of inertia, assuming a hollow tube with thickness t, can be written as a function of z using Equation 2.7.

$$I_y(z) = \frac{\pi}{64} d^4 - \frac{\pi}{64} (d - 2t)^4 \tag{2.8}$$

$$I_y(z) = \frac{\pi \left(d_b - \frac{z\,(d_b - d_t)}{z_0} \right)^4}{64} - \frac{\pi \left(2\,t - d_b + \frac{z\,(d_b - d_t)}{z_0} \right)^4}{64} \tag{2.9}$$

The deflection of the beam can be calculated using the Euler–Bernoulli equation:

$$\frac{d^2 y_d}{dz^2} = -\frac{M_x}{E\, I_y(z)} \tag{2.10}$$

$$\frac{d^2 y_d}{dz^2} = -\frac{F_y\, z - F_y\, z_0}{E \left(\frac{\pi \left(d_b - \frac{z\,(d_b - d_t)}{z_0} \right)^4}{64} - \frac{\pi \left(2\,t - d_b + \frac{z\,(d_b - d_t)}{z_0} \right)^4}{64} \right)} \tag{2.11}$$

The slope of the beam, θ_y, can be calculated by integrating Equation 2.11:

$$\theta_y = \frac{dy_d}{dz} = \int -\frac{F_y\, z - F_y\, z_0}{E \left(\frac{\pi \left(d_b - \frac{z\,(d_b - d_t)}{z_0} \right)^4}{64} - \frac{\pi \left(2\,t - d_b + \frac{z\,(d_b - d_t)}{z_0} \right)^4}{64} \right)} \, dz \tag{2.12}$$

Integrating the equation above results in the following equation for θ_y:

$$\theta_y = \frac{\ln\left(d_b\, z - d_t\, z - d_b\, z_0 + t\, z_0\right)\ \left(8\, F_y\, d_t\, {z_0}^2 - 8\, F_y\, t\, {z_0}^2\right)}{E\,\pi\,{d_b}^2\, t^3 - 2\,E\,\pi\, d_b\, d_t\, t^3 + E\,\pi\,{d_t}^2\, t^3}$$
$$- \frac{4\,\ln\left(d_b\, z - d_t\, z - d_b\, z_0 + t\, z_0\ (1-\mathrm{i})\right)\ \left(F_y\, d_t\, {z_0}^2 + F_y\, t\, {z_0}^2\ (-1+1\mathrm{i})\right)}{E\,\pi\,{d_b}^2\, t^3 - 2\,E\,\pi\, d_b\, d_t\, t^3 + E\,\pi\,{d_t}^2\, t^3}$$
$$- \frac{4\,\ln\left(d_b\, z - d_t\, z - d_b\, z_0 + t\, z_0\ (1+1\mathrm{i})\right)\ \left(F_y\, d_t\, {z_0}^2 + F_y\, t\, {z_0}^2\ (-1-\mathrm{i})\right)}{E\,\pi\,{d_b}^2\, t^3 - 2\,E\,\pi\, d_b\, d_t\, t^3 + E\,\pi\,{d_t}^2\, t^3} + C_{y1} \tag{2.13}$$

The tower is fixed at the base (no translation or rotation about the x-axis), so the slope at $z = 0$ is $\theta = 0$. Using this boundary condition, C_{y1} can be determined:

$$C_{y1} = -\frac{4\,F_y\,{z_0}^2\,Q_1(z)}{E\,t^3\,\pi\left(d_b - d_t\right)^2} \tag{2.14}$$

where,

$$\begin{aligned} Q_1(z) = {} & 2\,d_t\ \ln\left(-z_0\ \left(d_b - t\right)\right) - d_t\ \ln\left(-z_0\ \left(d_b + t\ (-1-\mathrm{i})\right)\right) \\ & - d_t\ \ln\left(-z_0\ \left(d_b + t\ (-1+1\mathrm{i})\right)\right) - 2\,t\ \ln\left(-z_0\ \left(d_b - t\right)\right) \\ & + t\ \ln\left(-z_0\ \left(d_b + t\ (-1-\mathrm{i})\right)\right)\ (1+1\mathrm{i}) \\ & + t\ \ln\left(-z_0\ \left(d_b + t\ (-1+1\mathrm{i})\right)\right)\ (1-\mathrm{i}) \end{aligned}$$

This results in the following slope equation:

$$\theta_y = \frac{4\,F_y\,{z_0}^2\,Q_2(z)}{E\,t^3\,\pi\left(d_b - d_t\right)^2} + C_{y1} \tag{2.15}$$

where,

$$
\begin{aligned}
Q_2(z) = {} & 2\,d_t\,\ln\left(d_b\,z - d_t\,z - d_b\,z_0 + t\,z_0\right) \\
& - d_t\,\ln\left(d_b\,z - d_t\,z - d_b\,z_0 + t\,z_0\,\left(1 - \mathrm{i}\right)\right) \\
& - d_t\,\ln\left(d_b\,z - d_t\,z - d_b\,z_0 + t\,z_0\,\left(1 + 1\mathrm{i}\right)\right) \\
& - 2\,d_t\,\ln\left(-z_0\,\left(d_b - t\right)\right) + d_t\,\ln\left(-z_0\,\left(d_b + t\,\left(-1 - \mathrm{i}\right)\right)\right) \\
& + d_t\,\ln\left(-z_0\,\left(d_b + t\,\left(-1 + 1\mathrm{i}\right)\right)\right) \\
& - 2\,t\,\ln\left(d_b\,z - d_t\,z - d_b\,z_0 + t\,z_0\right) \\
& + t\,\ln\left(d_b\,z - d_t\,z - d_b\,z_0 + t\,z_0\,\left(1 - \mathrm{i}\right)\right)\,\left(1 - \mathrm{i}\right) \\
& + t\,\ln\left(d_b\,z - d_t\,z - d_b\,z_0 + t\,z_0\,\left(1 + 1\mathrm{i}\right)\right)\,\left(1 + 1\mathrm{i}\right) \\
& + 2\,t\,\ln\left(-z_0\,\left(d_b - t\right)\right) + t\,\ln\left(-z_0\,\left(d_b + t\,\left(-1 - \mathrm{i}\right)\right)\right)\,\left(-1 - \mathrm{i}\right) \\
& + t\,\ln\left(-z_0\,\left(d_b + t\,\left(-1 + 1\mathrm{i}\right)\right)\right)\,\left(-1 + 1\mathrm{i}\right)
\end{aligned}
$$

At this point, it becomes clear that an analytic solution for the spring constant is not practical. Integrating the slope equation (Equation 2.15) would produce the deflection equation; however this integral becomes extremely complicated. A cleaner option is to calculate the spring constants numerically.

2.2.2.2 Numerical calculation of the y-direction spring constant, K_y

To determine the spring constant numerically, a vector of z values must be created to be used in the numerical integration.

To calculate the slope of the tower, θ, the curvature equation needs to be numerically integrated:

$$\theta(z) = \int_0^{z_0} \frac{-M(z)}{E\,I(z)}\,dz \tag{2.16}$$

The applied force, F_y, in the moment equation (Equation 2.6) can be factored out because it is not a function of z. This results in the following equation for the slope:

$$\theta(z) = \frac{F_y}{E} \int_0^{z_0} \frac{z_0 - z}{I(z)}\, dz \tag{2.17}$$

The integral in Equation 2.17 can be approximated by a sum of n elements and calculated numerically.

$$\theta_n(z) = \frac{F_y}{E} \sum_{i=0}^{n} \frac{z_0 - z_i}{I(z_i)}\, \Delta z_i \tag{2.18}$$

Equation 2.18 is a much simpler method for calculating the beam slope compared to the analytic solution shown in Equation 2.15. The deflection of the tower, y_d, can be calculated by integrating the slope equation.

$$y_d = \int_0^{z_0} \theta(z)\, dz = \int_0^{z_0} \left(\frac{F_y}{E} \sum_{i=0}^{n} \frac{z_0 - z_i}{I(z_i)}\, \Delta z_i \right) dz \tag{2.19}$$

The above equation can be numerically approximated with a summation of m elements as follows:

$$y_d = \sum_{j=0}^{m} \left(\frac{F_y}{E} \sum_{i=0}^{n} \frac{z_0 - z_{ij}}{I(z_{ij})}\, \Delta z_i \right) \Delta z_j \tag{2.20}$$

$$y_d = \frac{F_y}{E} \sum_{j=0}^{m} \sum_{i=0}^{n} \frac{z_0 - z_{ij}}{I(z_{ij})}\, \Delta z_i \Delta z_j \tag{2.21}$$

To determine the spring constant, the deflection equation should be rewritten in the

Table 2.2: Calculated spring constant and natural frequency for the y-direction vibration.

Parameter	**Tapered Beam Model (This paper)**	**FEA value (Gwon [17])**	**Modal Analysis (Katsanis [15])**
f_x	0.80 Hz	0.83 Hz	0.81 Hz
f_y	0.59 Hz	0.58 Hz	0.59 Hz

form, $F = -kx$, which in this case is:

$$F_y = -k_y\, y_d \tag{2.22}$$

$$F_y = \frac{E}{\sum_{j=0}^{m} \sum_{i=0}^{n} \frac{z_{ij}-z_0}{I(z_{ij})}\, \Delta z_i \Delta z_j}\, y_d \tag{2.23}$$

From Equation 2.23, is can be seen that the spring constant is:

$$k_y = \frac{E}{\sum_{j=0}^{m} \sum_{i=0}^{n} \frac{z_{ij}-z_0}{I(z_{ij})}\, \Delta z_i \Delta z_j} \tag{2.24}$$

Table 2.2 shows the calculated spring constant values and natural frequencies in the y-direction for the turbine parameters. The constant top area value is the spring constant calculated assuming a uniform circular cross-section beam with a profile equal to the top of the tower. The constant bottom area value is the spring constant calculated assuming a uniform circular cross-section beam with a profile equal to the bottom of the tower. The FEA values are derived in Tom Gwon's analysis [17], although it is unclear whether these values apply for the y or x direction. The tapered beam calculation is performed in MATLAB using the following code (with turbine parameters listed in Table 2.3):

```
z = linspace(0, z_0, 1000); % [m] Create the vector of z values
M_F = (z - z_0); % [m] The moment equation with respect to the ...
    applied force (M/F)
d_F = d_b - z*(d_b-d_t)/z_0; % [m] The diameter as a function of ...
    height
I = pi/64*d.^4 - (pi/64)*(d-2*t).^4; % [m^4] The moment of inertia
v_F = -M_F ./ (E*I); % [s^2/m^2/kg] The curvature equation
theta_F = cumtrapz(z, v_F); % [s^2/m/kg] The slope equation
y_F = cumtrapz(z, theta_F); % [m/N] The displacement with respect ...
    to force
ky = 1 ./ y_F(end); % [N/m] Spring constant
```

The intermediate variables from the MATLAB calculation are shown in Figure 2.2. The curvature, slope, and displacement are all with respect to force, since it cancels out of the equation and is not used. The spring constant is calculated by taking the inverse of the displacement/force curve at the largest z value. These plots were created using a z-vector of 1000 elements.

Table 2.3: Turbine parameters used in the MATLAB analysis for calculating k_y

Variable	Value	Description
z_0	70 ft	The total height of the tower
d_b	20 in	The diameter of the bottom of the tower
d_t	8.7 in	The diameter of the top of the tower
E	30 Mpsi	The modulus of elasticity for ASTM A572 Grade-50 Steel
t	0.2 in	The thickness of the turbine tower tube

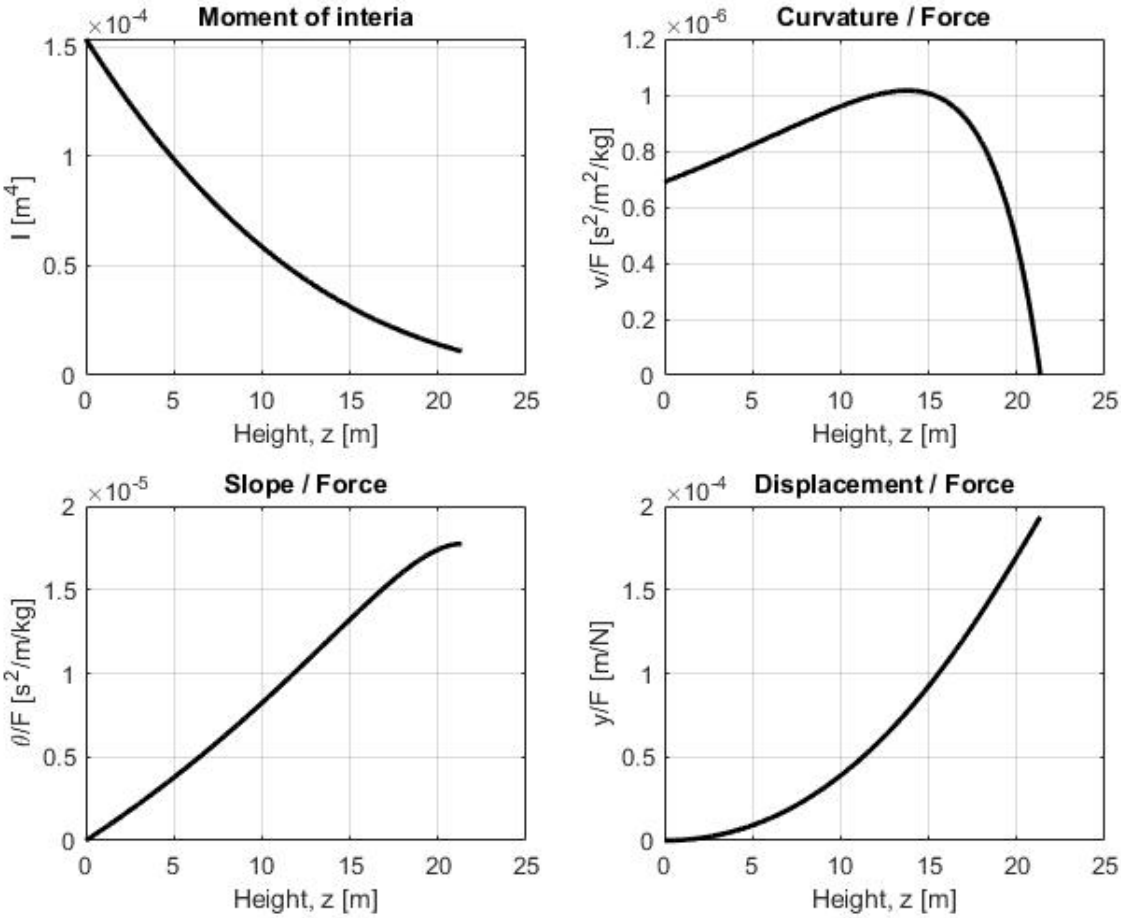

Figure 2.2: The intermediate variables from the numerical spring constant calculation in MATLAB.

2.2.3 Determining Effective Mass

The cantilever beam (Figure 2.3) can be simplified to a massless beam with all of the mass concentrated at the end of the beam.

As shown in Figure 2.2, the deflection at any point on the beam, $y(z)$ can be calculated numerically. The normalized deflection, $y_n(z)$ can be calculated by dividing this equation by the maximum deflection at the free end, z_0 (Equation 2.25). This allows the deflection equation to be rewritten in terms of the normalized deflection (Equation 2.26).

$$y_n(z) = \frac{y(z)}{y(z_0)} \tag{2.25}$$

$$y(z) = y_n(z) \cdot y(z_0) = y_n(z) \cdot y_{max} \tag{2.26}$$

$$\tag{2.27}$$

The velocity of the beam at point z can be calculated by taking the derivative of the deflection equation (Equation 2.26). The velocity of the beam is shown in Equation 2.28.

$$v(z) = y_n(z) \cdot v_{max} \tag{2.28}$$

The kinetic energy, dE_k, of the beam at a single point, x, can be calculated as shown

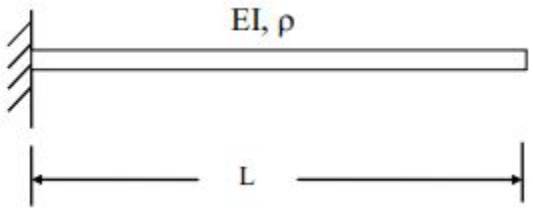

Figure 2.3: A cantilever beam model [24]

in Equation 2.29.

$$dE_k = \frac{1}{2}mv^2$$
$$dE_k = \frac{1}{2}\left(\rho A(z)dz\right)\left(y_n(z)v_{max}\right)^2 \tag{2.29}$$

Using Equation 2.29, the total kinetic energy of the beam can be found by integrating dE_k, as shown in Equation 2.32.

$$E_k = \int_0^{z_0} dE_k \tag{2.30}$$
$$E_k = \int_0^{z_0} \left(\frac{1}{2}\rho A(z)\left(y_n(z)v_{max}\right)^2 dz\right) \tag{2.31}$$
$$E_k = \frac{1}{2}\rho v_{max}^2 \int_0^{z_0} \left(A(z)y_n^2(z)dz\right) \tag{2.32}$$

The effective mass of the beam can be calculated by examining Equation 2.32 and comparing to the standard format, $E = \frac{1}{2}mv^2$. Equation 2.33 shows the resulting effective mass.

$$m_{eff} = \rho \int_0^{z_0} \left(A(z)y_n^2(z)dz\right) \tag{2.33}$$

The deflection from Equation 2.21 can be substituted into Equation 2.33. Additionally, the integral can be converted to a numerical sum as shown in Equation 2.34.

$$m_{eff} = \rho \sum_{k=0}^{p} \left(A(z_p)y_n^2(z_p)\Delta z\right) \tag{2.34}$$

Numerically calculating m_{eff} using MATLAB, results in a value of $m_{eff} = 140$ [kg].

The total effective mass at the top of the tower, $m_{eff,total}$ is as follows:

$$m_{eff,total} = m_{nacelle} + m_{eff} \tag{2.35}$$
$$m_{eff,total} = 209 \text{ [kg]} + 170 \text{ [kg]} = 379 \text{ [kg]} \tag{2.36}$$

The mass in Equation 2.36 is the lumped sum used to calculate the parameters in Table 2.2 and create the turbine tower model.

2.2.4 Developing State Space Equations

Vibrations in the x and y directions can be split up into 2 independent 2nd order differential equations:

$$\ddot{x} + \frac{C_x}{m}\dot{x} + \frac{k_x}{m}x = \frac{1}{m}F_x(t) \tag{2.37}$$

$$\ddot{y} + \frac{C_y}{m}\dot{y} + \frac{k_y}{m}x = \frac{1}{m}F_y(t) \tag{2.38}$$

These equations can be converted to state space form:

$$\dot{\boldsymbol{x}} = \boldsymbol{A_x}\boldsymbol{x} + \boldsymbol{B_x}F_x(t) \tag{2.39}$$

$$x = \boldsymbol{C_x}\boldsymbol{x} + \boldsymbol{D_x}F_x(t) \tag{2.40}$$

$$\dot{\boldsymbol{y}} = \boldsymbol{A_y}\boldsymbol{y} + \boldsymbol{B_y}F_y(t) \tag{2.41}$$

$$y = \boldsymbol{C_y}\boldsymbol{y} + \boldsymbol{D_y}F_y(t) \tag{2.42}$$

Where the full equations are written out as follows:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ -\frac{k_x}{m} & -\frac{C_x}{m}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}0\\ \frac{1}{m}\end{bmatrix}\cdot F_x(t) \tag{2.43}$$

$$x = \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + 0\cdot F_x(t) \tag{2.44}$$

$$\begin{bmatrix}\dot{y}_1\\ \dot{y}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ -\frac{k_y}{m} & -\frac{C_y}{m}\end{bmatrix}\begin{bmatrix}y_1\\ y_2\end{bmatrix} + \begin{bmatrix}0\\ \frac{1}{m}\end{bmatrix}\cdot F_y(t) \tag{2.45}$$

$$y = \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}y_1\\ y_2\end{bmatrix} + 0\cdot F_y(t) \tag{2.46}$$

These equations are for a simple cantilever model with a lumped mass at the end of the beam (lumped mass accounts for nacelle and tower mass). The resulting natural

frequency from this model matches the ABAQUS FEA results from Gwon's paper [17]. A comparison between the resulting natural frequency from this model and the ABAQUS FEA results in Gwon's analysis [17] can be seen in Table 2.2.

Chapter 3

DIGITAL SIGNAL PROCESSING MODEL

3.1 Model Block Diagram

The blade imbalance detection is simulated using MATLAB/Simulink. Figure 3.1 shows the block diagram for a method of imbalance detection where the rotor frequency is known. This is an accurate method, but requires the information from an encoder on the shaft or an accurate phase-locked loop (PLL) controller.

The force due to an imbalance is given in Equation 2.1. It is a periodic function with the same frequency as the rotor and is proportional to the effective imbalance mass, m. The variable forcing function is the input for the lumped parameter turbine tower state space model. These vibrations are then read by an acceleration sensor with a sampling rate of 100 Hz. The frequency of interest is demodulated using a lock-in amplifier or DFT, which should result in the acceleration from only the imbalance. The acceleration at the top of the tower should have higher amplitude vibrations at the rotor frequency when there is an imbalance in the blades (causing an eccentric mass).

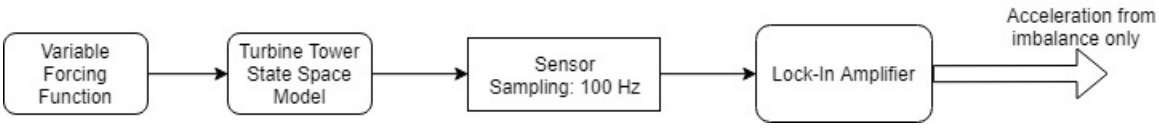

Figure 3.1: A block diagram showing the lock-in amplifier and the turbine tower simulation. This figure was created with Draw.io.

3.2 Processing the data

This paper describes a few ways to process the acceleration data from the turbine tower. One method relies on accurate frequency information to precisely demodulate the signal at the rotor frequency. An alternative method is to lock onto the strong frequency vibration of the tower using a phased-lock loop controller. Finally, a discrete Fourier transform can be calculated to transform the time domain data to the frequency domain. There are various ways to make this process more efficient, including zoom FFTs and the Goertzel algorithm.

The simulation and turbine tower state space model produce displacements shown in Figure 3.2. The same simulation data is shown in the frequency domain in Figure 3.3. From the frequency domain, it can be seen that there is resonance at 0.84 Hz (the natural frequency of the tower in the x-direction). Additionally, there is an excitation at the rotor frequency that has a magnitude proportional to the amount of mass at the end of the blade. This simplified linear model produces an displacement signal that can be represented by the sum of sines as shown in Equation 3.1 (this can be seen in Figure 3.2).

$$y = C_1 \sin(\omega_n t + \phi_1) + C_2 \sin(\omega_{drive} t + \phi_2) \tag{3.1}$$

The first method for processing the acceleration data utilizes a lock-in amplifier to translate the data in the frequency domain. This new data, with the rotor frequency shifted to 0 Hz, is filtered and results in a DC acceleration that should be proportional to the imbalance mass. The second method utilizes a zoom FFT, which provides higher resolution than a typical FFT in the bandwidth of interest. The third method uses a Goertzel algorithm to calculate the bin of interest in the DFT.

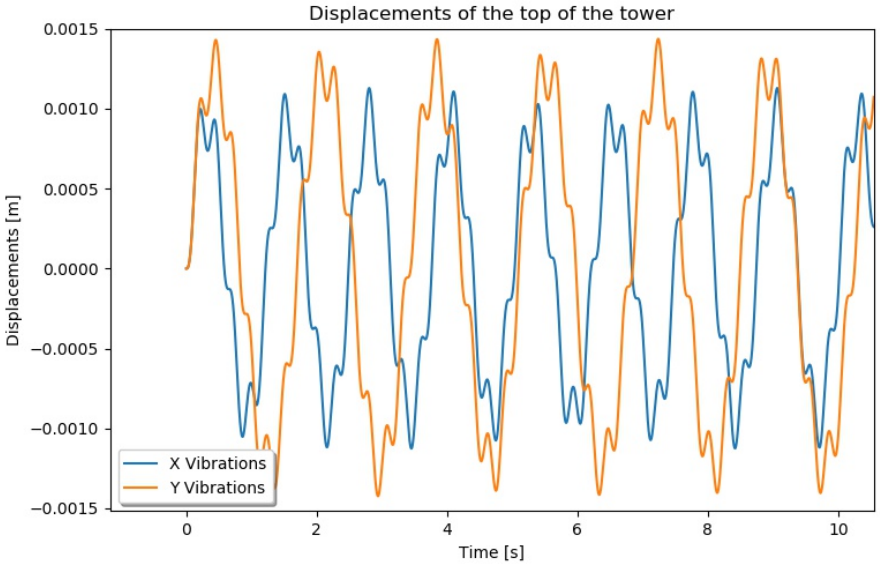

Figure 3.2: The simulated displacement for a rotor imbalance at 230 RPM. The imbalance mass is placed at the end of the blade, which causes accelerations and displacements at the top of the tower. This figure was created with Python.

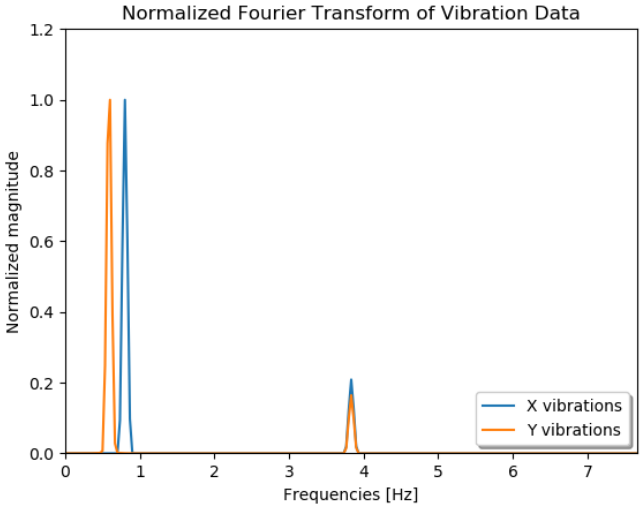

Figure 3.3: This figure shows the simulated displacement data in the frequency domain. The rotor frequency is 230 RPM in this simulation. This figure was created with Python.

3.2.1 DFT review

A Discrete Fourier Transform (DFT) converts a signal from the time domain to the frequency domain. Typically, this is performed with using the Fast Fourier Transform (FFT) algorithm, which is an efficient method for calculating the DFT. The DFT is defined by the formula in Equation 3.2 [37].

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{3.2}$$

$\vec{x}$ is the original signal in the time domain with N elements, and $\vec{X}$ is the signal in the frequency domain with N elements. Equation 3.2 can be rewritten using Euler's formula to show DFT equation in terms of sine and cosine values (this is important for understanding the lock-in amplifier in a future section). Equation 3.3 shows the new expression for the DFT equation.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right)\right] \tag{3.3}$$

From Equation 3.3, it can be seen that the DFT can be calculated by mixing the signal, $\vec{x}$, with a sine and cosine signal and summing the result for each frequency. The DFT equation can be further rewritten using vector notation as in Equation 3.7 and defining some reference vectors. This equation shows that the DFT of a signal is an imaginary vector that can be created from the dot products of the original signal

and some reference sine and cosine signals.

$$\vec{n}_r = \begin{bmatrix} 0/N \\ 1/N \\ 2/N \\ \vdots \\ (N-1)/N \end{bmatrix} \tag{3.4}$$

$$\vec{r}_{sin} = \sin\left(2\pi k \vec{n}_r\right) \tag{3.5}$$

$$\vec{r}_{cos} = \cos\left(2\pi k \vec{n}_r\right) \tag{3.6}$$

$$X_k = \vec{x}^T \cdot \vec{r}_{cos} - j\vec{x}^T \cdot \vec{r}_{sin} \tag{3.7}$$

Another way of looking at the DFT (from the form in Equation 3.7) is as a time domain convolution with a low pass filter. The summation (shown explicitly in the form from Equation 3.3) acts as a simple finite impulse response (FIR) decimation filter with unity coefficients and a decimation rate of $d = N$, commonly known as an averaging filter. When thinking about the DFT in this way, some possible optimization methods start to become clear. For example, we may not need to calculate the frequency component for every kth element in $\vec{X}$, and we may choose to use a better filter that has stronger attenuation at higher frequencies. This is the basis for the lock-in amplifier design described in the next section.

3.2.2 Lock-In Amplifier

A lock-in amplifier is used to measure AC signals in particularly noisy environments (Figure 3.4). It works by multiplying the noisy signal by a reference signal created by an internal oscillator. By using two reference signals out of phase, the real and imaginary parts of the signal at the specified carrier frequency can be calculated [25].

Let's say there is an unknown sinusoidal signal with high frequency noise ($\omega_2 >> \omega_1$):

$$y = A\,\sin(\omega_1\,t) + B\,\sin(\omega_2\,t) \tag{3.8}$$

The amplitude of the signal, A, can be determined by multiplying the signal, y, by a reference signal with the same frequency as ω_1 and filtering:

$$\begin{aligned} y \cdot \sin(\omega_1\,t) &= [A\,\sin(\omega_1\,t) + B\,\sin(\omega_2\,t)]\,\sin(\omega_1\,t) && (3.9)\\ &= A\,\sin^2(\omega_1\,t) + B\,\sin(\omega_1\,t)\,\sin(\omega_2\,t) && (3.10)\\ &= A\left(\frac{1}{2} - \frac{1}{2}\,\cos(2\,\omega_1\,t)\right) + B\,\sin(\omega_1\,t)\,\sin(\omega_2\,t) && (3.11) \end{aligned}$$

Note that Equation 3.9 resembles the imaginary part of the standard DFT equation (Equation 3.3), where the input signal is multiplied by a sine wave at a set reference frequency.

A filter is applied to Equation 3.11 (similar to the summation filter used in Equation 3.3), with a cutoff frequency that is smaller than ω_1. This results in all terms approaching zero, except for $\frac{A}{2}$, which can be calculated and solved for A to achieve the magnitude of the frequency component at ω_1. To calculate the magnitude and phase of the signal y, it must be multiplied by a reference sine signal (as in Equation 3.9) and a reference cosine signal. The following equation shows the result of the mixing step (phase-sensitive detection) in the lock-in amplifier:

$$Y = X\cos(\omega t) + jX\sin(\omega t) \tag{3.12}$$

where X is the noisy AC signal and Y is the signal that is fed into a filter to remove any unwanted frequency components at ω_1 and $2\omega_1$ and produce a signal, $Y_{filtered}$ that only contains the DC component:

$$Y_{filtered} = \frac{A_{real}}{2} + j\frac{A_{imaginary}}{2} \tag{3.13}$$

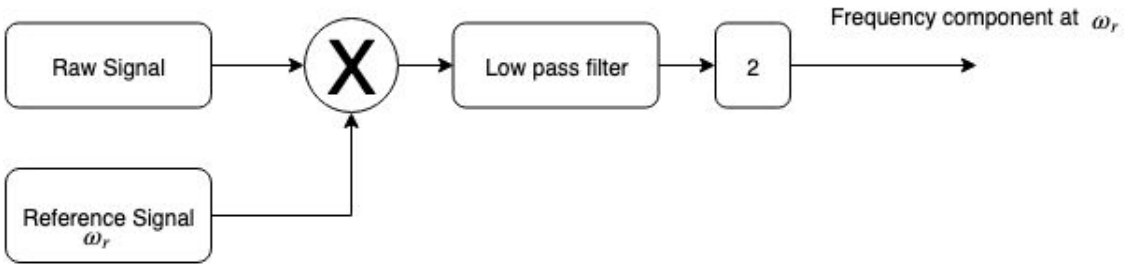

Figure 3.4: A block diagram showing the details of a lock-in amplifier. This figure was created with Draw.io.

Using the lock in amplifier tuned to a constant frequency, Figure 3.5 shows the resulting output. This shows that an imbalance can be detected by measuring the output of a lock-in amplifier that is tuned to the frequency of the turbine rotor. When the acceleration data is treated as an AM signal, with the carrier frequency equal to the rotor frequency, the acceleration due to the rotation of the blades can be identified. This acceleration is directly related to the mass at the end of the blade, and should be a constant value (as long as the rotor frequency is accurately known).

Lock-in amplifiers (LIAs) are typically used when the frequency is relatively stable and the signal-to-noise ratio is low. For this application, the rotor frequency can change, which may not give the LIA filters enough time to settle. If there is a very stable and accurate rotor speed controller, this method would be ideal because it extracts the signal of interest despite the high noise and distortions in the measurement.

3.2.2.1 Types of low-pass filters

The filter choice is very important for a lock-in amplifier. Some of the common options are finite impulse response (FIR) filters, and infinite impulse response (IIR) filters, and the Savitzky-Golay (SG) filter.

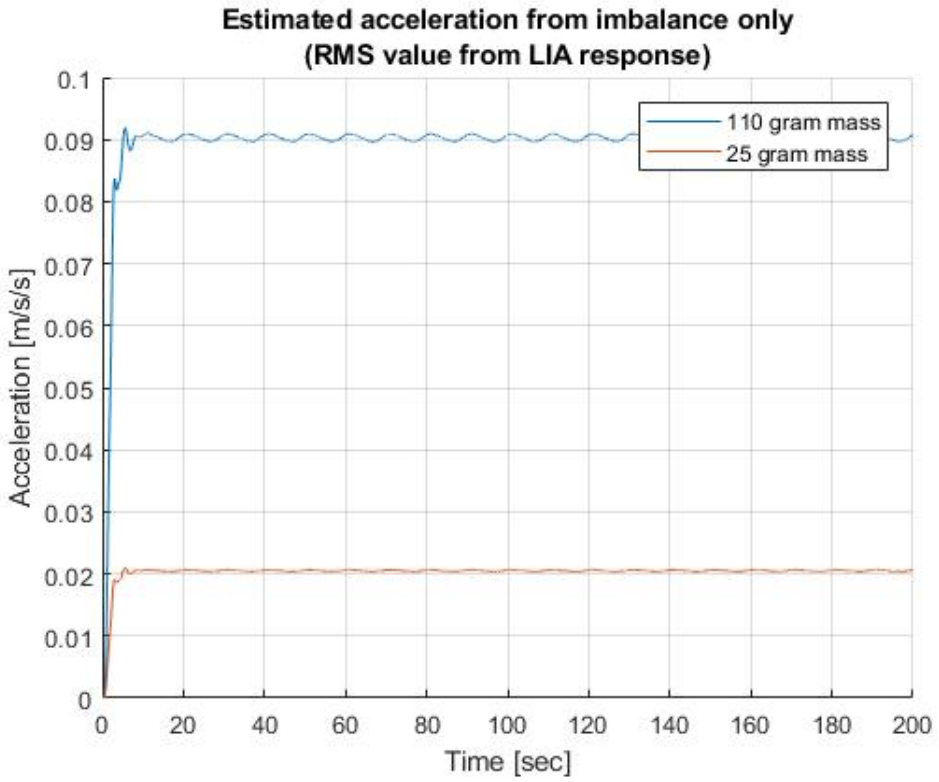

Figure 3.5: The output of the lock-in amplifier from the simulated data shown in Figure 3.2. This figure was created with Python.

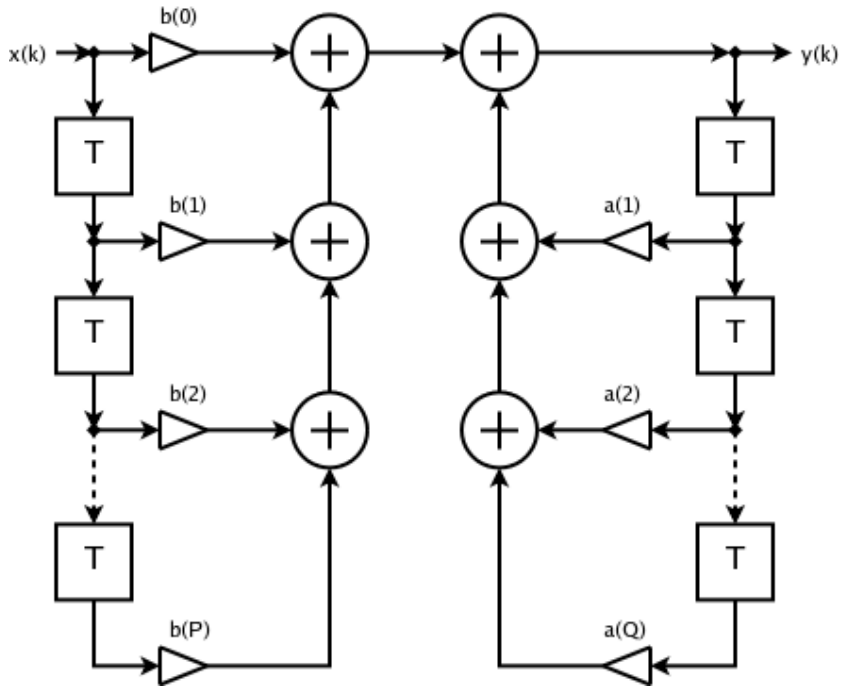

Figure 3.6: A block diagram of a standard IIR filter. [36]

The IIR filter is shown in Figure 3.6, which significantly smooths the data, but at the cost of some time-domain distortion. An IIR filter uses past states in the output calculation, meaning an impulse input signal will affect the filter output forever (*infinite* impulse response filter). This results in a transfer function as follows:

$$T(z) = \frac{b(z)}{a(z)} \tag{3.14}$$

Figure 3.7 shows the result of a Butterworth IIR filter compared to a moving average filter. This shows that a customized filter is much more effective than standard moving average methods of smoothing data.

Another filter alternative is the FIR filter, which does not rely on past states to calculate the filter output. This results in a finite response to an impulse input and a

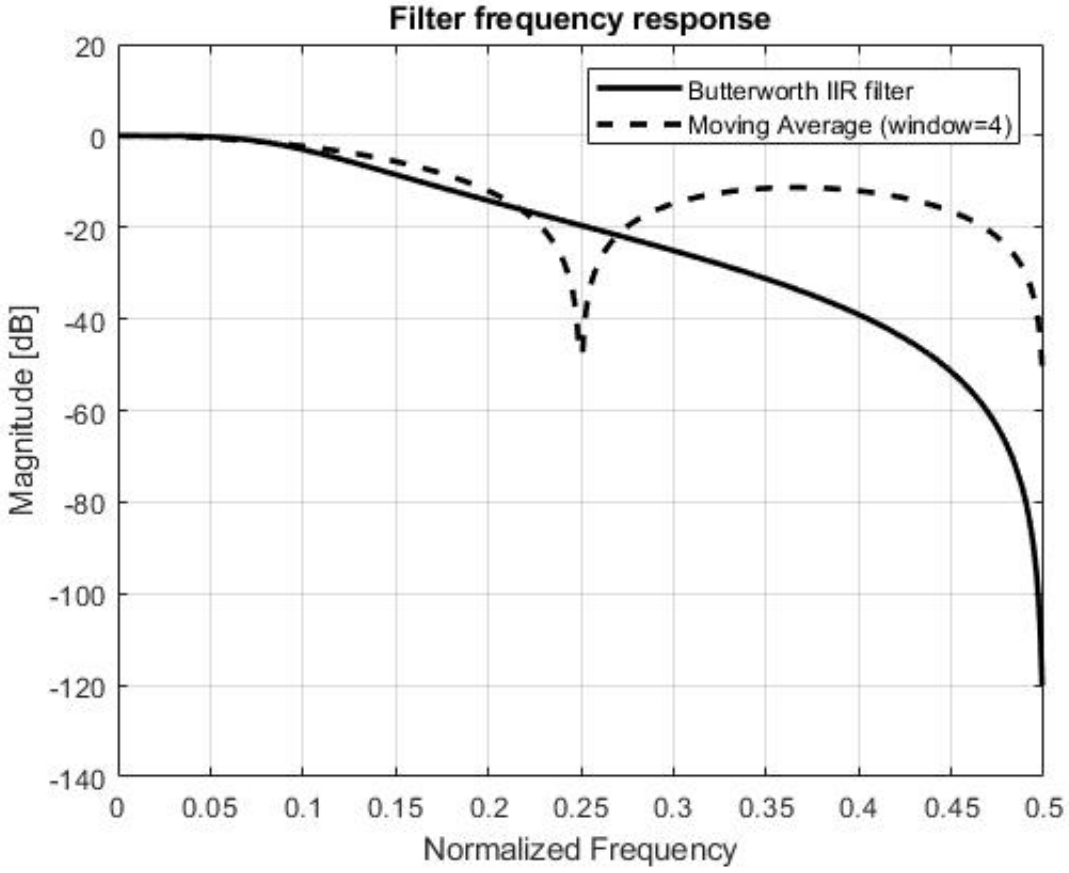

Figure 3.7: The filter response of the Butterworth IIR filter (Equation 3.14) compared to a moving average filter with a window size of 4. This figure was created in MATLAB with 2 simulated filter designs.

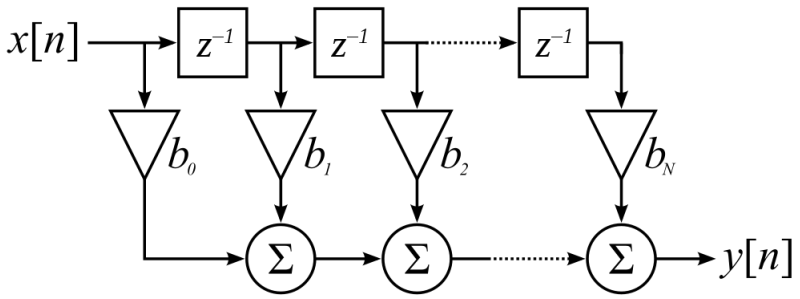

Figure 3.8: A block diagram of a standard FIR filter. [35]

linear phase. A common FIR filter that is used for smoothing data is a moving average (where all of the filter coefficients are identical). To achieve a more customized filter for this data, a FIR filter is designed using the Window method, which designs a FIR filter using a specified window. Figure 3.8 shows a block diagram of a FIR filter. One of the most common uses of FIR filters is to decimate a signal (not having to calculate previous states allows for some significant optimizations when decimation is involved). Because the past states are not used, the transfer function has a denominator of 1:

$$T(z) = \frac{b(z)}{1} \tag{3.15}$$

Another filter option for this data is the Savitzky-Golay (SG) filter, which increases the signal-to-noise ratio with out significantly distorting the signal. The SG filter uses convolution to fit a polynomial on a sliding window to the data. The output of this filter can be calculated using the following equation [29]:

$$Y_j = (\mathbf{C} \otimes y)_j = \sum_{i=-\frac{m-1}{2}}^{\frac{m-1}{2}} C_i y_{j+i} \tag{3.16}$$

C_i are the convolution coefficients (m total coefficients), y is the sampled data point, and j is the index of the data point.

The convolution coefficients can be selected from a table, but it is much more robust to calculate them prior to performing any filtering. The convolution coefficients, $\mathbf{C}$, are calculated using a variable change. The time vector for a given window, $\vec{x}$, is converted to $\vec{z}$ using the time step (Δt) with the following equation:

$$z = \frac{\vec{x} - \bar{x}}{\Delta t} \tag{3.17}$$

$\bar{x}$ is the time value at the central point of the window. $\vec{z}$ is used to calculate the matrix $\mathbf{J}$, where each $i-$throw of $\mathbf{J}$ has the values $1, z_i, z_i^2, z_i^3, ..., z_i^n$. After $\mathbf{J}$ has been created, the convolution coefficients can be calculated as follows:

$$\mathbf{C} = \left(\mathbf{J^T J}\right)^{-1} \mathbf{J^T} \tag{3.18}$$

The filtered data can then be calculated as the convolution of $\mathbf{C}$ and the unfiltered data (Equation 3.16).

3.2.3 Identifying the rotor frequency from the tower accelerations

If the rotor frequency cannot be measured directly, it is possible to obtain the frequency indirectly from the acceleration data. A phase-locked loop is a control system that minimizes the phase between to signals. As the phase error is driven to zero, the output signal frequency approaches the input signal's frequency. Phase-locked loops (PLLs) are commonly used in telecommunications where the frequency of a noisy signal can be recreated with an internal oscillator that locks on to the frequency of

the noisy signal. Figure 3.9 shows a block diagram of the PLL implementation with tower acceleration data as the input.

The first stage of the frequency detection is a narrow bandpass filter to get rid of high frequency noise and natural frequency resonance of the tower. This is multiplied by an internal oscillator that starts at an arbitrary frequency (a guess that is close to the actual frequency, but not necessarily accurate). This signal is passed through a lowpass filter to provide a steady phase error. The phase error is driven to 0 using a PID controller. This outputs the correct frequency of the signal that is then fed back and multiplied with the input signal.

One of the problems with a phase-locked loop is that the PID controller gains depend on the amplitude of the signal, which changes depending on the amount of imbalance and rotor frequency. The simplest approach to this would be to normalize the accelerations with the expected frequency. This, however, is not perfect and doesn't account for the acceleration variation due to different eccentric masses. A better solution would be to use a simple envelope detector to determine the amplitude of signal. The accelerations can be divided by the output of the envelope detector to normalize the signal. Figure 3.11 shows an example of a simple envelope detector[3]. The low-pass filter in the envelope detector can be designed based on the desired accuracy. For high computational efficiency, an averaging filter can be used because it minimizes the amount of multiply operations, and reduces the filter calculation to summations only.

3.2.4 Windowed FFT

One problem with standard FFTs is that the frequency content can get spread across all of the bins if the signal contains frequencies that do not fall exactly into any one

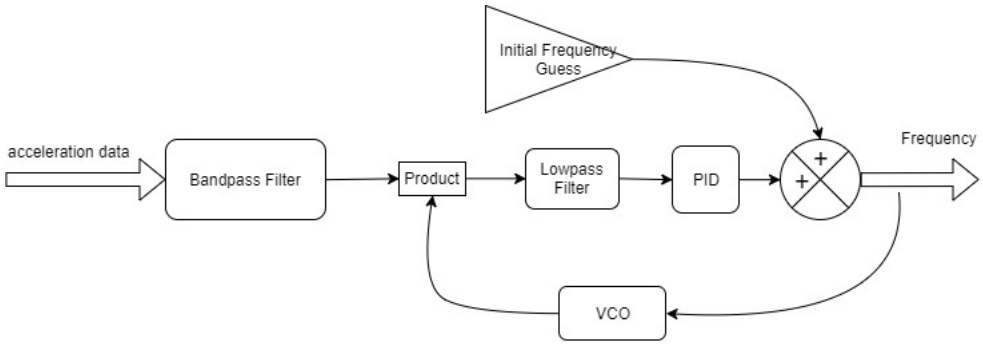

Figure 3.9: The block diagram of a phase-locked loop implementation when detecting rotor frequencies from acceleration measurements [21]. This diagram was created with Draw.io [4].

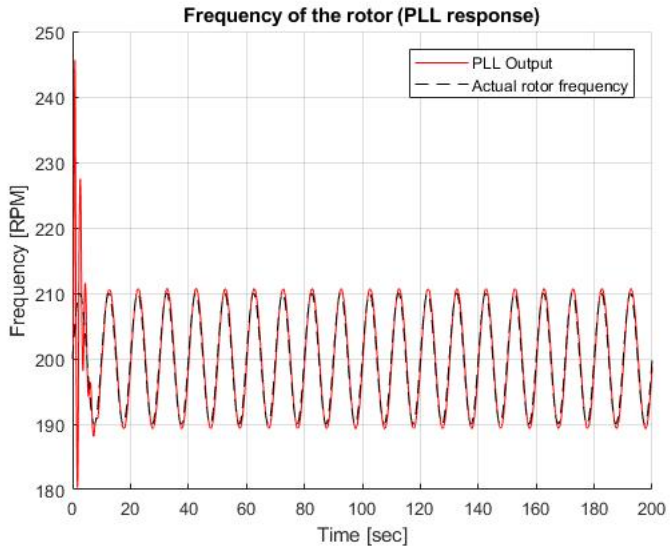

Figure 3.10: The output of the PLL when detecting frequency from the data shown in Figure 3.2. The frequency of the input data oscillates from 190 RPM to 210 RPM to simulate a real turbine controller that may oscillate between frequencies. This figure was created with MATLAB.

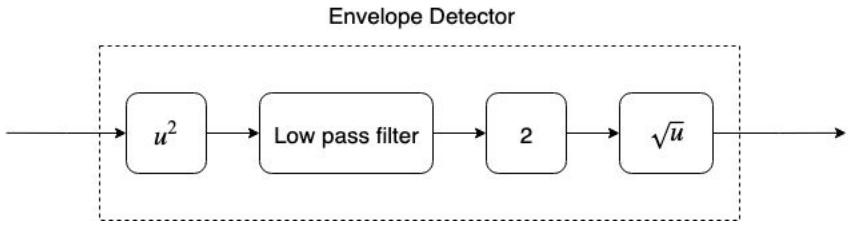

Figure 3.11: A block diagram for a simple envelope detector. The signal is first squared and multiplied by a gain of 2. This signal (which should be entirely positive) is sent through a lowpass filter to remove the high frequency information. The lowpass filter needs to reject $2f$, which is the resulting frequency when a signal with frequency, f is squared. The square root of the resulting signal produces the the amplitude of the signal. This figure was created with Draw.io.

bin. For example, a 1024-point FFT of a signal sampled at 128 Hz will have bins at each integer frequency value (0,1,2,3, ..., 64 Hz). If the measured signal contains a signal at 10.l Hz, the frequency spectrum will spread this component across all of the bins. This is shown by the standard FFT in Figure 3.12. In order to prevent this from happening, a windowing filter can be applied to the raw signal before the FFT is calculated. These windowing filters are weighted averages of the signal (rather than a uniform average that is performed by the standard FFT). Two of the most common windowing filters are the Blackman and Hamming windows. For FFT analysis, the Blackman filter is preferable because it has a narrower peak and rejects some of the higher frequencies more than the Hamming window. Figure 3.12 shows the result of a standard FFT and a windowed FFT applied to the same signal.

To apply a windowed FFT, the coefficients ($\vec{w}$) of the Blackman (or other windowing function) function must be created. The Blackman coefficients can be created analytically in Equation 3.19 [23]. In this equation, N is the length of the windowing filter (which is equal to the number of points in the FFT).

$$\vec{w}(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right), \quad 0 \leq n \leq (N-1) \qquad (3.19)$$

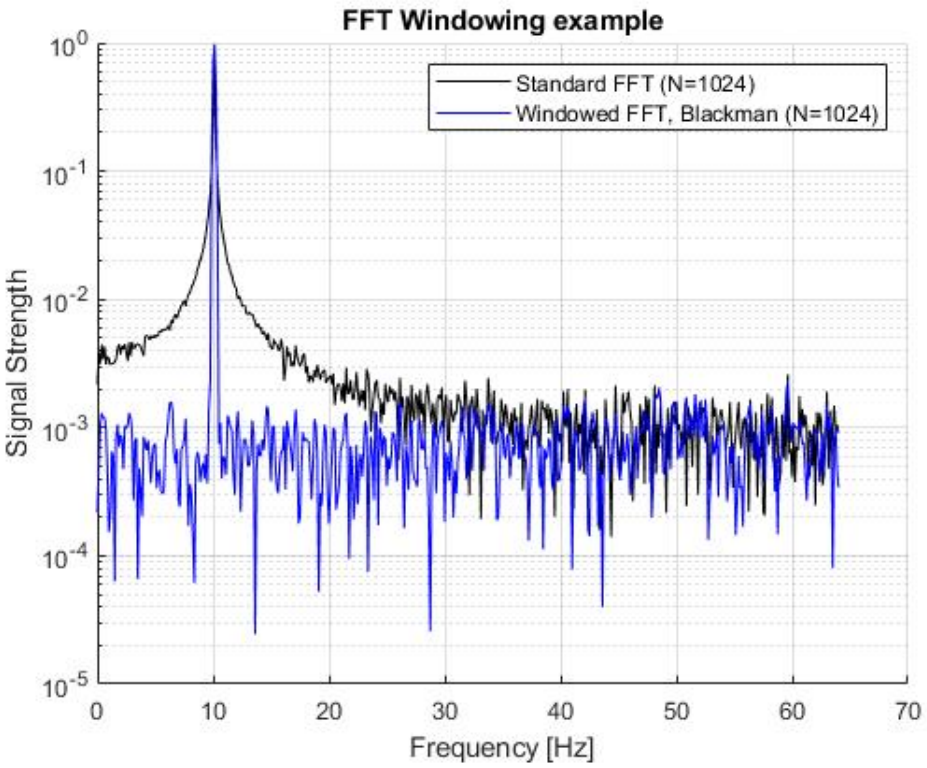

Figure 3.12: This figure shows a standard FFT and a windowed FFT with a sampling rate of 128 Hz. The FFT length is 1024, and the windowing filter is a Blackman filter. This figure was created with MATLAB.

The Blackman coefficients are then element-wise multiplied by the signal ($\vec{y}$) before the FFT is calculated as shown in Equation 3.21 ($\odot$ is the element-wise multiplication operator).

$$\vec{y}_{window} = \vec{y} \odot \vec{w} \tag{3.20}$$

$$m_{fft} = \texttt{FFT}\left(\vec{y}_{window}\right) \tag{3.21}$$

3.2.5 Zoom FFT

A very powerful method for reducing the computational cost of the FFT calculation is to use a Zoom FFT. If the imbalance detection device contains a low-power processor,

it will be able to run for a long time off of a battery. This also means that the processor could be much slower than an alternative high-power processor. A Zoom FFT will significantly reduce the computational load on the processor, allowing the low-power processor to sufficiently calculate the frequency spectrum of the input signal.

One of the main tradeoffs for the reduced computational load, is a reduced frequency spectrum range. For example, a set of 1024 data points sampled at F_s, will produce a frequency spectrum from 0 to $F_s/2$ using the standard FFT. A zoom FFT with a decimation rate of $D = 10$ can produce a frequency spectrum with a range of $F_s/20$ at 1/10th the computational cost.

If the frequency of the rotor is not precisely known, then a lock-in amplifier is difficult to implement. In order to detect excitation at the rotor frequency, the time-domain data can be transformed into the frequency domain. A discrete Fourier transform (DFT) is the frequency domain representation of a sampled signal; however, it can be slow and difficult to calculate on a microcontroller.

To efficiently calculate the Fourier transform, the DFT matrix can be factored and the complexity reduced from $O(n^2)$ to $O(n \log n)$. This efficient algorithm is called the fast Fourier transform (FFT). Despite being much more efficient than the standard DFT, a generic FFT is still inefficient for this application because most of the frequency data occurs outside the bandwidth of interest and is thrown away. This can be fixed by applying a zoom FFT algorithm.

The zoom FFT consists of 4 main steps. First the data is convoluted with a reference signal in the time domain (translated in the frequency domain) to shift a center frequency, F_c, to 0 Hz. Second, a lowpass filter prevents aliasing when sampling at a lower sample rate. An infinite impulse response (IIR) filter can be used for simplicity; however, the phase information of the signal is lost. To solve this, a finite impulse

response (FIR) filter can be used. FIR filters are more complicated but most have a flat phase response, ensuring the filter output retains the original phase information. Third, the data is decimated (re-sampled at a lower rate). Finally, the decimated data is passed through a FFT algorithm that produces a frequency spectrum in the specified bandwidth. Since the frequency range of the zoom FFT is much smaller, the resolution can be much higher for a FFT length. Alternatively, a much smaller FFT length can be used to produce the same resolution.

For a microcontroller sampling the acceleration of the turbine tower at 128 Hz, Figure 3.13 shows a block diagram of a zoom FFT implementation for detecting an imbalance. Point A represents the raw input signal, which is the acceleration of the tower sampled at $F_s = 128$ Hz in this case. The frequency translation is performed by mixing the input signal with a reference signal at F_c. The goal of a Zoom FFT is to reduce the amount of data by decimating the original signal and reducing the sampling rate. Just performing a sampling rate reduction would result in aliasing, so there needs to be a digital anti-aliasing filter that prepares the signal for the decimation step. This digital filter is a low pass FIR filter that produces a filtered signal at Point C, which can then be decimated without aliasing to achieve Point D. This new set of data is sampled at $F_{s,new} = \frac{F_s}{D}$, which means the frequency spectrum spans a much smaller range.

Zoom FFTs [10] contain the following parameters:

F_c: The frequency center (shifted to 0 Hz by the zoom FFT)

BW: The bandwidth of interest

F_s: The sampling rate

N: The length of the FFT

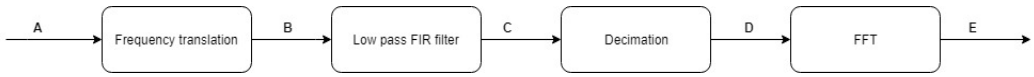

Figure 3.13: A block diagram showing the zoom FFT algorithm. This figure was created with Draw.io.

The decimation factor, D, is calculated as follows:

$$D = floor\left(\frac{F_s}{BW}\right)$$

Where the length, L, of the initial buffer into the decimator is:

$$L = D \cdot N$$

The computational reduction for a Zoom FFT is shown in Figure 3.14. However, it is important to note that a stronger filter is required for higher decimation factors, D. This means there is a trade-off between FFT computational load and filter computational load. A zoom FFT is just the name of a method of frequency transformation where the data is decimated before applying the FFT algorithm. The computational savings occur when a well-designed lowpass filter does not negate the workload reduction caused by the FFT. Essentially, this is a much more flexible FFT where different aspects can be modified to optimize for the application. In this case, the Zoom FFT can provide a much higher resolution frequency analysis because the frequencies of interest are much smaller than the sampling rate.

3.2.5.1 Frequency translation

Shifting data in the frequency domain is a common technique used in AM radio signals. Audio frequencies are typically mixed with a high frequency carrier that, according to the Nyquist criteria, would require an extremely high sample rate to capture the signal.

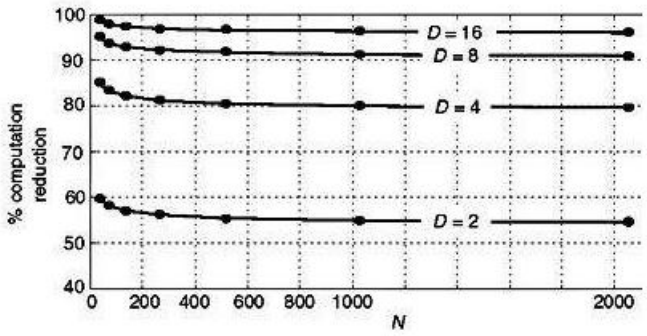

Figure 3.14: A diagram showing the percent computational workload reduction of a $\frac{N}{D}$-point Zoom FFT relative to a standard n-points FFT [21].

If ω is the rotor frequency, multiplying the time domain signal by $\cos(-\omega t)$ is the same as convolving the data in the frequency domain. Since the frequency spectra for $\cos(-\omega t)$ is symmetric across the y-axis, the time-domain must be multiplied by an imaginary wave.

The equation for the frequency-shifted time-domain data is:

$$Y = Xe^{-i(\omega t)} \tag{3.22}$$

where Y is the shifted data, X is the original data, ω is the rotor frequency, and t is the time. Based on Equation 3.22, the real and imaginary parts of the shifted signal are shown below [10]:

$$Y_{real} = X\cos(\omega t) \tag{3.23}$$

$$Y_{imaginary} = -X\sin(\omega t) \tag{3.24}$$

An example of frequency translation is shown in Figure 3.15. The blue signal is the frequency-shifted signal, which has a higher frequency at 2 times the original signal frequency. In order to remove the extra component at 10 Hz, the digital filter (shown with a dotted blue line) must be applied to the shifted signal.

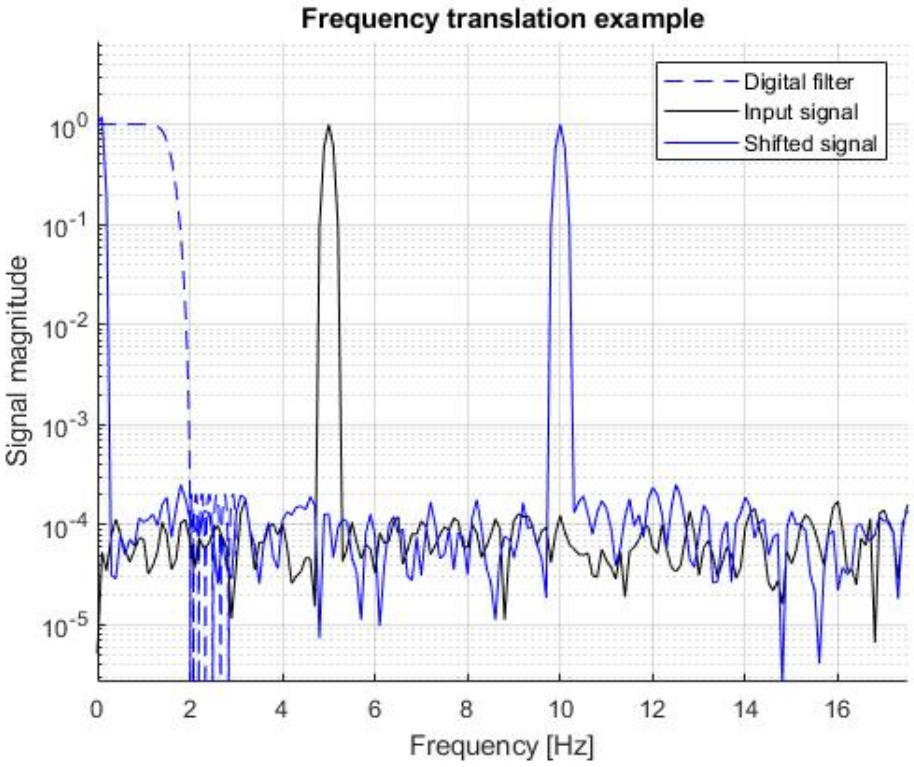

Figure 3.15: This figure shows an example of frequency translation. The black signal is the frequency spectrum of some input signal that has a strong component at 5 Hz. The blue signal shows the frequency shifted spectrum that results from multiplying the input signal by a reference signal with a frequency of 5 Hz (as shown in Equation 3.24). The dotted blue line shows the digital filter response. This figure was generated with MATLAB.

3.2.5.2 Decimation/downsampling

Decimation is the process of reducing the sample rate of a signal. Before the data is downsampled, the high frequency components are reduced with a lowpass filter. This filter is also called an anti-aliasing filter because it prevents high frequency data from being misinterpreted at a different sample rate. Equation 3.25 shows the equation used for calculating the output of a FIR decimator, where x is the input data, h is the impulse response, K is the length, and D is the decimation factor[19].

$$y[n] = \sum_{k=0}^{K-1} x[nD - k] \cdot h[k] \tag{3.25}$$

Figure 3.15 shows a visual representation of the decimation process and anti-aliasing filter. The shifted signal (blue) contains important frequency information in the range 0 - 2 Hz. Any higher frequencies have been significantly attenuated by the digital filter (dotted blue line). This means the blue data can be decimated to a new sampling rate of 4 Hz, without any aliasing effects (because the digital filter ensures there are no strong components above 2 Hz).

3.2.5.3 An optimized Zoom FFT implementation

Table 3.1 shows the parameters for the Zoom FFT. These values have been strategically picked to make the optimization process easier. In general, some nice optimizations can be achieved if the number of filter coefficients ($N_b + 1$) is an integer multiple of the sampling rate (F_s).

In Table 3.1, the FFT length is the number of samples in the buffer used as the input to the FFT. An FFT length of 1024 means that 1024 samples are used as an input to the FFT algorithm. The FIR filter order is 255, which is high because it needs to strongly attenuate higher frequencies with a narrow pass band. Despite having

Table 3.1: Zoom FFT parameters

Variable	Value	Description
F_{min}	2 Hz	The minimum frequency for the Zoom FFT calculation
F_s	128 Hz	The original sampling rate
F_r	4 Hz	The frequency range of the Zoom FFT
N_{fft}	128	FFT length (buffer length used in the FFT calculation)
N_b	255	FIR filter order
D	$\frac{F_s}{2F_r} = 16$	Decimation rate

an extremely large order, the FIR filter is simple and computationally inexpensive to implement (this will be shown later in this section). An order of $N_b = 255$ means the filter order will have 256 ($N_B + 1$) coefficients. F_{min} and F_r define the frequency range that the Zoom FFT is targeting. More specifically, the Zoom FFT will calculate the frequency spectrum of the input signal only from F_{min} to F_r. The filter design is highly motivated by the frequency range because it needs to attenuate the original signal sufficiently (to prevent aliasing at the decimation step) at the edge of the frequency range.

The reference signal for the Zoom FFT is created at the minimum frequency, F_{min} shown in Table 3.1 and is shown in Equation 3.26.

$$y_{ref} = \sin\left(F_{min} \cdot 2\pi \cdot time\right) \tag{3.26}$$

Note that there are only 64 unique values in a reference signal with the specified F_{min}. This means that all of these values can be precomputed at initialization and stored in memory. This removes the $\sin()$ operations from the real-time calculations, which reduces the computational time, since trig functions are one of the slowest floating-point operations (they are typically about 15X slower than a floating-point addition

operation) [6].

One of the benefits for using a finite impulse response (FIR) filter is that there are no recursive calculations. This produces a very important property of FIR filters, which is that not all the outputs need to be calculated. Assuming there are $(N_b + 1)$ filter coefficients, a random window of (N_b+1) samples can be properly filtered at any time during operation. If we are decimating the input signal, this means we do not have to calculate the filter outputs for samples that are removed due to the decimation step.

The down-sampled data, y_{ds}, can be calculated as follows ($\odot$ is the element-wise multiplication operator):

$$y_{ds} = \vec{y}^T \cdot \left(\vec{y}_{ref} \odot 2\vec{b}\right) \tag{3.27}$$

In Equation 3.27, y_{ds} is the value at the decimated sampling rate. $\vec{y}$ is a sliding buffer of the original input data, which is a vector of $(N_b + 1)$ values. $\vec{y}_{ref}$ is the vector of the reference signal values, which also has a length of $(N_b + 1)$. $\vec{b}$ is the vector of FIR filter coefficients. The factor of 2 in this equation is a result of the derivation of the frequency-shift process. Equation 3.27 is calculated at the decimated sampling rate (F_{sD}), which is shown in Equation 3.28.

$$F_{sD} = \frac{F_s}{D} \tag{3.28}$$

Equation 3.27 can also be used with larger filter orders. Effectively, this means that the filtering and decimation part of the Zoom FFT can be performed with just a single dot product and an element-wise vector multiplication. Using the values in Table 3.1, the following steps can be followed to create a Zoom FFT algorithm:

1. Precompute the unique values of the reference signal during initialization.

2. Design a FIR filter with an appropriate cutoff frequency and filter order. The

cutoff frequency (F_{cutoff}) is equal to the frequency range (F_r) of the Zoom FFT. The coefficients of this filter are stored in $\vec{b}$.

3. Generate a sliding buffer of data with a window size of $(N_b + 1)$ and an overlap of $(N_b + 1 - \frac{F_s}{D})$, which is stored in $\vec{y}$.

4. Perform the calculation shown in Equation 3.27.

5. Calculate the FFT of the newly decimated data.

Figure 3.16 shows the results of a MATLAB simulation using the Zoom FFT to calculate the frequency spectrum. In this simulation, the standard FFT uses an FFT length of 2048, while the Zoom FFT only has a length of $N_{fft} = 128$. This is a significant reduction in computation for the same frequency resolution in the target Zoom FFT range. Both a standard FFT and a Zoom FFT use the same FFT algorithm, so the difference in performance comes from the lower sampling rate, and therefore smaller frequency range.

3.2.5.4 Estimating processor performance

Comparing the computational time of different algorithms is difficult because the dominant factor is usually memory access, rather than actual floating-point operations [27]. Evaluating the computational cost due to memory access patterns in complicated, so this analysis will only consider the computational cost of the floating-point operations (FLOPs). To minimize the cost from memory access patterns, the algorithms will be vectorized and converted to matrix form. Most processors have highly optimized matrix operation functions, which should hopefully make arithmetic the dominant factor when determining the computational cost.

For this analysis, a STM32 processor is used as the baseline for estimating cycle

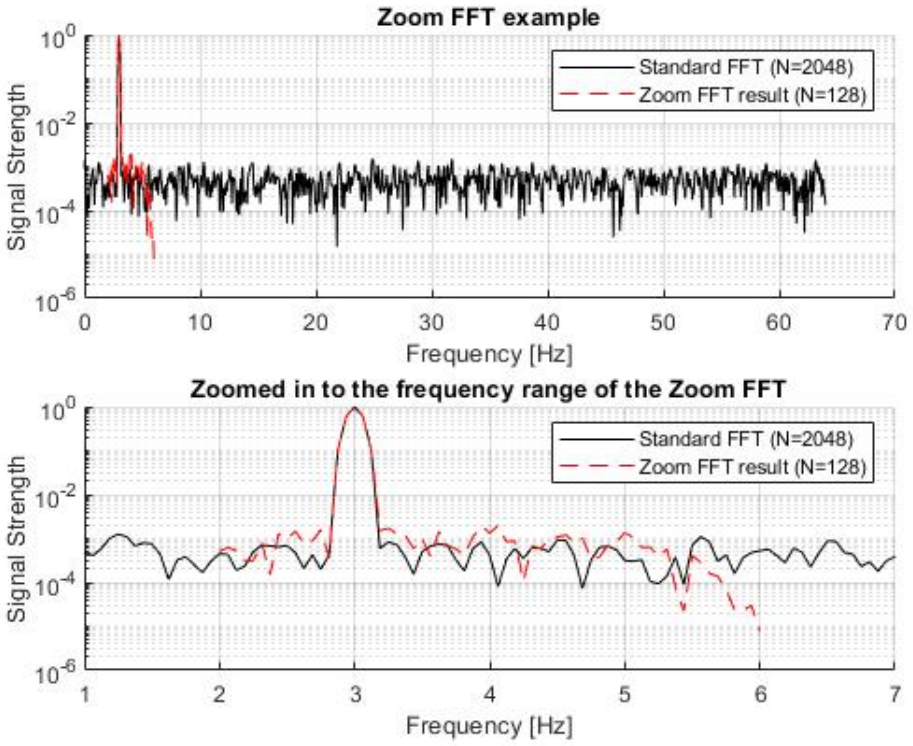

Figure 3.16: This figure shows the results of a Zoom FFT using the parameters listed in Table 3.1. The top plot shows the frequency spectrum from a standard FFT (black) and a Zoom FFT (red). The bottom plot shows the same data as the top plot, but zoomed into a narrow frequency range. The standard FFT has a length of 2048, while the Zoom FFT has a length of 128. This figure was created with MATLAB.

counts because it is the unit used to prototype the LifeLine data acquisition device that collects experimental acceleration data from the turbine. The STM32 has its own single precision register set to handle floating-point operations with hardware (rather than using compiled C libraries to perform the floating-point operations in many complex steps). The floating-point unit (FPU) in the STM32 offers arithmetic instructions for the operations listed in Table 3.2 [33].

Many FPUs (like the one in Table 3.2), have multiply/accumulate arithmetic instructions, which means they can perform a multiplication and addition all in one step. Additionally, division operations are much more expensive than multiplication operations and should be avoided as much as possible when designing algorithms.

Using the estimated floating point performance, the estimated clock cycles are shown in Table 3.3. These are very rough estimates of the computational power required to perform each of the 3 FFT techniques. The number of clock cycles for the FFT calculation is estimated from the Cooley Tukey algorithm [30]. The windowed FFT (shown in Figure 3.12) is not significantly more computationally expensive than a standard FFT, but performs much better. The Zoom FFT analyzed in this table is the same design from Table 3.1 and Figure 3.16.

3.2.6 Goertzel Algorithm

The Goertzel algorithm is a method for calculating individual bins of a discrete Fourier transform (DFT) without calculating all of the bins. Since imbalance detection depends on the strength of the rotor frequency, this algorithm produces the particular rotor frequency bin value without calculating the entire DFT. Fundamentally, this algorithm operates similarly to the Zoom FFT previously described, and is often called a Goertzel filter because the algorithm takes the form of a digital filter. This algo-

Table 3.2: Cycle counts for a STM32 floating-point processor. The exponential* operation is estimated from a floating-point performance analysis [6] because it is not stated in the STM32 datasheet.

Operation	Cycles
Absolute value	1
Negate of a float	1
Addition/subtraction	1
Multiply	3
Multiply and accumulate/subtract	3
Divide	14
Square Root	14
Exponential	20*

Table 3.3: Estimated cycle counts for various FFT techniques.

	Number of Operations		Number of Clock Cycles [kCycles]		
Method	**Multiply/ Accumulate**	**FFT Length**	**Multiply**	**FFT**	**Total**
Standard FFT	0	1024	0	435.2	435.2
Windowed FFT	1024	1024	3.072	435.2	438.3
Zoom FFT	512	128	1.536	38.1	39.6

rithm applies 2 stages of filtering, where the first stage is an infinite impulse response (IIR) filter, and the second stage is a finite impulse response (FIR) filter.

For the first stage of the Goertzel algorithm, a 2nd order IIR filter is applied to x:

$$s[n] = x[n] + 2\cos(\omega_0)s[n-1] - s[n-2] \tag{3.29}$$

This IIR filter follows a Direct Form II structure, where 2 state variables are required for a 2nd order filter. ω_0 is defined as the normalized frequency to be analyzed.

The second stage of the algorithm is a FIR filter, which is known for its linear phase, and lack of feedback. This means that the filter output does not depends on any previous outputs.

$$y[n] = s[n] - e^{-j\omega_0}s[n-1] \tag{3.30}$$

Equation 3.30 shows the output of the Goertzel equation, $y[n]$.

There is an optimized version of the Goertzel algorithm that is even faster than the general form, but it produces squared magnitude components at each frequency instead of real/imaginary pairs [8]. The phase of the frequency component is not used in the imbalance detection algorithm, so the optimized Goertzel algorithm is well suited for this application. One of the benefits of the Goertzel algorithm is the buffer lengths do not need to be a power of 2. An overview of the algorithm in a block diagram form is shown in Figure 3.17.

The Goertzel algorithm is simulated with MATLAB and the results are shown in Figure 3.18. The Goertzel algorithm calculates individual frequency components of the DFT, which are identical to the values calculated with a standard FFT algorithm.

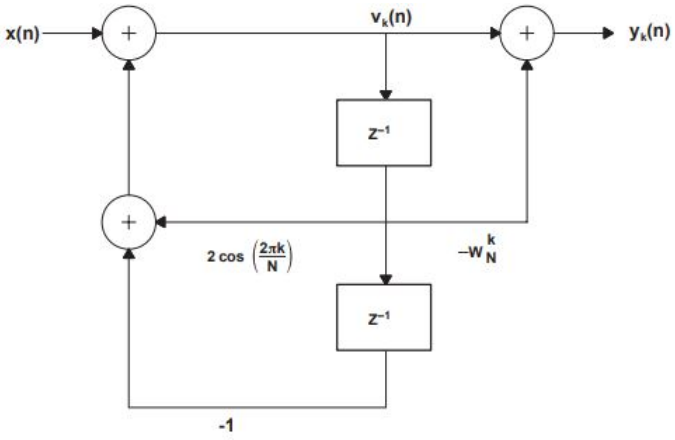

Figure 3.17: The direct-form realization of the Goertzel Algorithm[18].

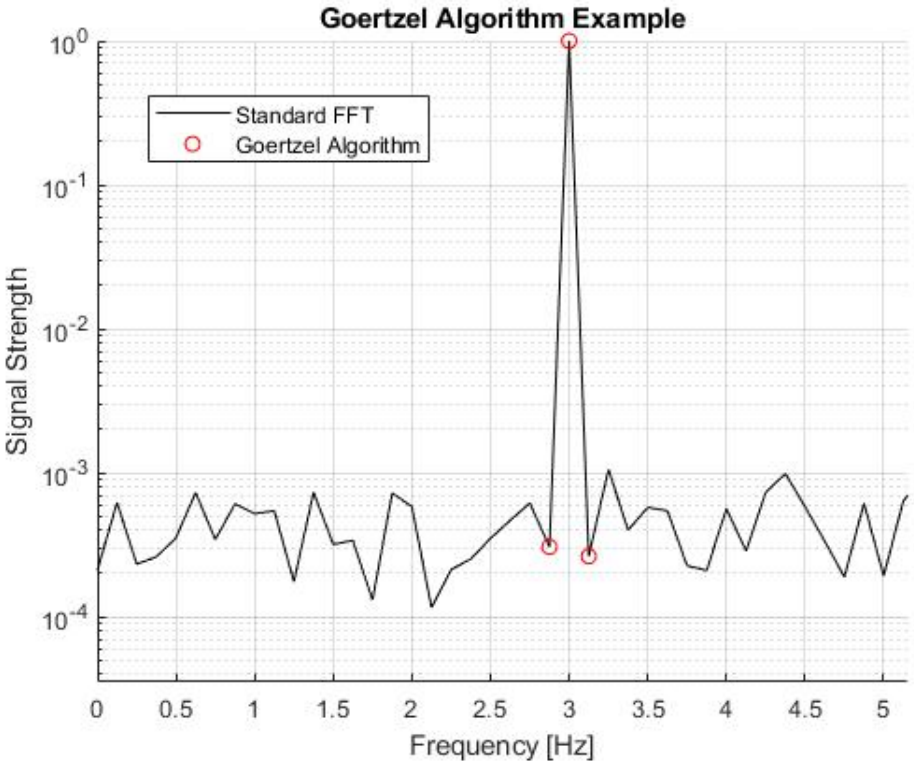

Figure 3.18: This figure shows the frequency spectrum calculated with a standard FFT (black) and a few points calculated with the Goertzel algorithm (red). This figure was created in MATLAB.

Chapter 4

IMBALANCE DETECTION ALGORITHM

4.1 Overview

A rotor imbalance can causes problems and failures with the turbine, so it is important to detect the "state" of the turbine using some measured input. Turbine dynamics are complicated, vary between different systems and can be dependent on many variables. An effective way to develop a model that is optimized for each different turbine is to use machine learning (ML). Machine learning is the study of algorithms that build mathematical models to perform specific tasks without explicit instructions.

4.1.1 Machine learning background

The 2 main types of machine learning algorithms are supervised and unsupervised learning. Supervised algorithms assume that each data set is assigned an output, while unsupervised algorithms take a data set of only inputs and attempt to create groups and classes from the input data. For this application, each measurement data set from the turbine will be manually labeled as "good" or "bad" (or as "balanced"/"unbalanced"), which is a type of supervised learning.

Machine learning algorithms also fall into the "regression" or "classification" categories. Regression algorithms are used if the output is a continuous function, while classification algorithms are used if the output falls into distinct classes. The output for the turbine detection algorithm is the state of the turbine (balanced/imbalanced), which is a discrete classification.

Knowing that the ML algorithm must be a classification, supervised model, the specific algorithm choice can be narrowed down. Some of the common algorithms are logistic regression (this can be used as a classification algorithm even though it is called "regression"), k-nearest neighbor, and neural networks. This paper will analyze each of these models and apply them to the experimental turbine data.

4.2 The training data

Typically, it is common for machine learning algorithms to split the data into test and training sub groups. This allows the accuracy of the models to be trained and evaluated on different data sets. The input data for the turbine is a 256-point frequency spectrum of the acceleration data at the top of the tower. This data is obtained using the LifeLine system, which includes an accelerometer and a STM32 microprocessor. This device is placed at the top of the tower and powered directly from the turbine. Ideally, this device would significantly drop the clock rate of the processor or swap out the processor for an extremely low-power one. This would allow the device to be wireless and have a reasonable operation time for each battery charge. Figure 4.1 shows a spectrogram of the measured acceleration data from the LifeLine device. A spectrogram is a way to visualize frequency data over time in a 2-dimensional figure, where the magnitudes of each frequency component are represented by a different color. The spectrogram shows that there is a strong frequency component at about 2.4 Hz (the rotor frequency) that is constant throughout most of the operation.

Currently, only a single set of about 100 training examples exists. The data can be separated into 2 classes; however, there is no information about which is the *balanced* and *imbalanced* classes. To properly train the machine learning models, more data is required. Despite the lack of available data, this paper will discuss the methods and

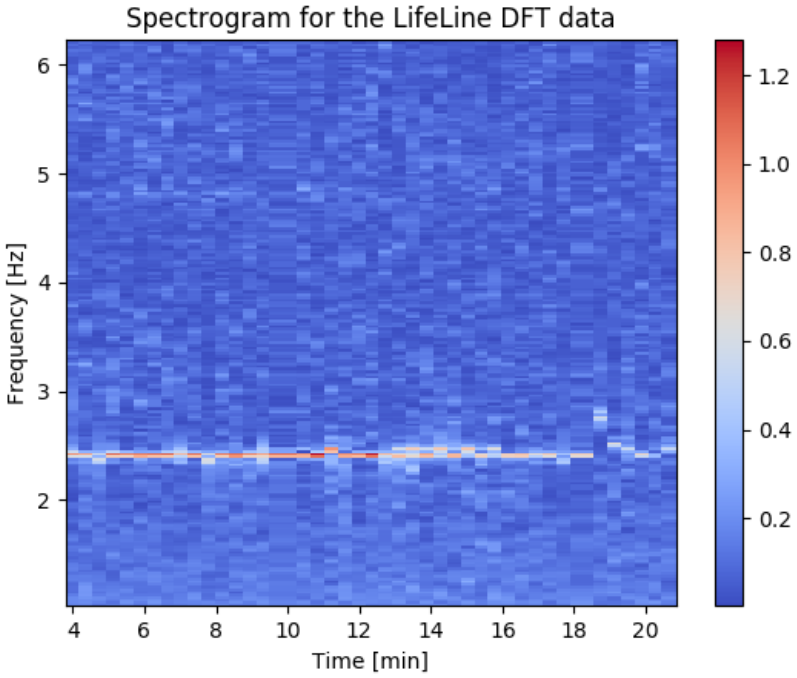

Figure 4.1: This figure shows the spectrogram of the tower accelerations. This figure was created with Python.

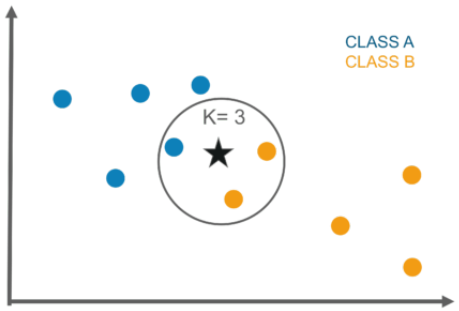

Figure 4.2: A visualization of the KNN algorithm [7]. The training data for a 2-class problem with 2 inputs is shown as blue and orange dots. For $K = 3$, a circle is drawn around the new point (star) until 3 training data points fall within the circle. The classification of the new point is then determined by the majority of the points inside the circle. In this case, there are more *Class B* points than *Class A* points, so the new point (star) will be classified as *Class B*.

implementations of each algorithm using the limited data set.

4.3 K-Nearest Neighbor algorithm

4.4 About the algorithm

The k-nearest neighbor (KNN) algorithm is one of the simplest models, but can be the most accurate and powerful when dealing with fairly small data sets. This model essentially "memorizes" the training data and plots the points in n-dimensional space. The classification of a new point is calculated based on the distances between the new point and the training points. For example, Figure 4.2 shows a visualization of the classification process for a 2-class system. Since this algorithm uses all of the training data in memory, it becomes very slow with many inputs, and many training data sets. The example in Figure 4.2 is a 2-dimensional problem (only 2 inputs) with only a few data points, so the KNN algorithm works very well.

The K-NN algorithm uses a "majority voting" method where the Euclidian distance between all the training data points and the test data is calculated. This model assumes there is a relatively equal amount of balanced rotor experimental data and imbalanced experimental rotor data, which means the sample weighting can be uniform. If there is a much higher frequency class (for example, much more balanced experimental data), this class will tend to dominate the calculations regardless of the actual class of the test data.

The number of neighbors, k, is chosen to be 3 for this algorithm (using trial and error). A larger value of k will reduce the classification noise, but the boundaries are much more general and not as tuned to the training data. A lower value of k will match the model very closely to the training data, but may add noise into the classification process.

The Euclidean distance is simply the distance between the data points. This equation is shown below:

$$d = \sqrt{\sum (x_{i,a} - x_{j,a})^2} \tag{4.1}$$

4.4.1 Choosing the data

The KNN algorithm works well on small data sets, so the DFT data should be pre-processed and cut down to a smaller, 2-dimensional data set. Based on a simplified dynamic model of the turbine tower, a rotor imbalance will cause a high frequency excitation at the rotor frequency. Using the maximum frequency component of the experimental DFT data and the corresponding frequency is a good way to capture enough information about the tower, while also reducing the data set from 256 input points to only 2 input points.

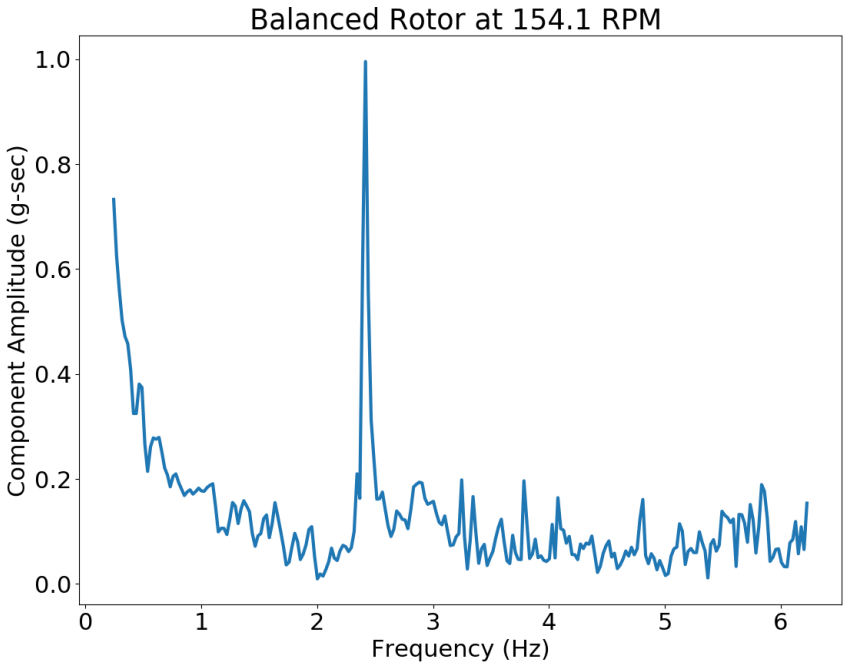

Figure 4.3: This figure shows a single DFT result (a single column from Figure 4.1). This figure was created with Python.

Figure 4.3 shows the frequency spectrum from a single DFT calculation. To convert this data into 2-dimensional data for the KNN algorithm, the peak value and corresponding frequency are extracted from this data. For example, Figure 4.3 would produce a 2-dimensional data set as shown in Equation 4.2.

$$[x_1, x_2, y] = [frequency, amplitude, class] = [2.3, 1.0, ClassA] \tag{4.2}$$

By converting the entire data set for all training examples into 2-dimensional data, the data in Figure 4.1 can be represented as the data in Figure 4.4.

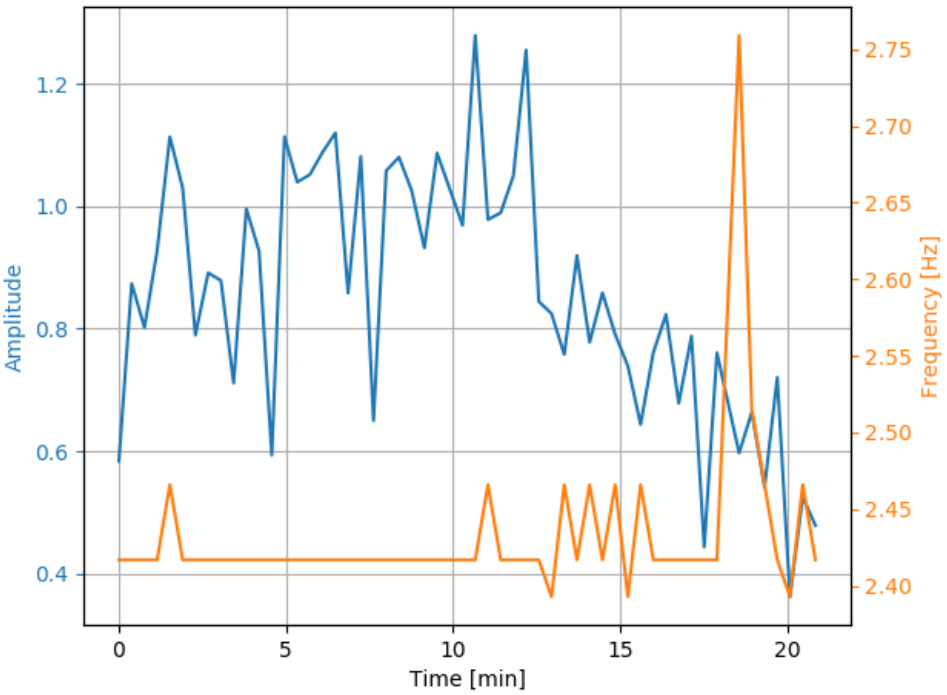

Figure 4.4: This figure shows a the 2-dimensional data used as an input to the KNN classification algorithm. This figure was created with Python.

Table 4.1: An example of the terminology for the confusion matrix in Equation 4.3. *Class A* and *Class B* represent the different classes of data. These can be related to a *balanced* and *imbalanced* rotor with proper data set labels.

	Predicted Class A	**Predicted Class B**
Actual Class A	12	1
Actual Class B	3	6

4.4.2 Algorithm Results

To apply the algorithm, the data is converted into short lists containing a maximum amplitude, a corresponding frequency value, and the known class (Equation 4.2). For this test, there are 55 experimental DFT results for *Class A* and 55 experimental DFT results for *Class B*.

Training the KNN model is extremely fast and only involves storing all of the training data in memory. When the algorithm is applied to new test examples, the results can be visualized in a confusion matrix [11] as shown in Equation 4.3. A confusion matrix is a common tool used to describe the performance of a classification algorithm. This is an n by n matrix (where n is the number of classes) that essentially shows the amount of correct and incorrect guesses for each class. Table 4.1 shows a labeled version of the confusion matrix shown in Equation 4.3.

$$C_{confusion} = \begin{bmatrix} 12 & 1 \\ 3 & 6 \end{bmatrix} \tag{4.3}$$

A nice way to visualize the results of the classification algorithm with 2-dimensional inputs is to create a decision boundary plot. A decision boundary is the area in

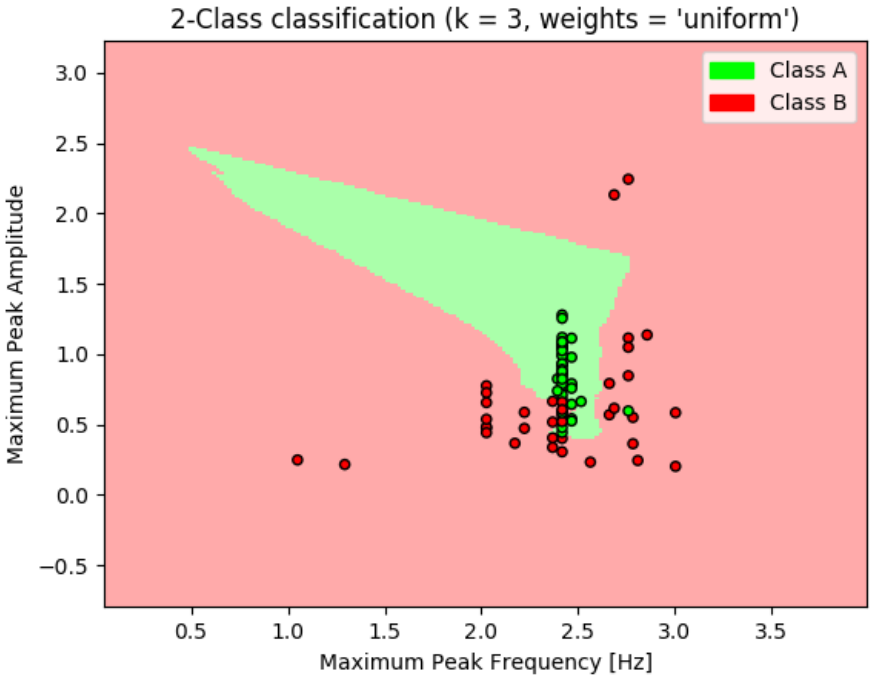

Figure 4.5: This figure shows the classification boundary for the KNN algorithm with $K = 3$ for all of the training data. This figure was created with Python.

space where the algorithm switches to a different classification. Figure 4.5 shows the classification boundary for the KNN algorithm with $K = 3$.

Initially, the balanced and imbalanced classes were expected to be easily distinguishable from each other. Despite not having accurate labels of the experimental data, the imbalanced rotor data was expected to have much higher amplitudes at the rotor frequency; however, from Figure 4.5, there isn't a significant difference between the amplitudes of the 2 data classes.

One observation that can be made from the data shown in Figure 4.5 is that the Class A data seems to have more frequency stability. The Class A data has a constant frequency of about 2.5 Hz, while the Class B data has varying frequencies. There is not enough data to make any hard conclusions, but this variance could be caused by either wind speed fluctuations or an actual property of the imbalance. It is possible that an imbalance in the rotor could cause more variations in rotor speed, despite not having significant amplitude differences. However, until the data can be accurately labeled, or more experimental data is collected, these classes should remain Class A and Class B to avoid any incorrect assumptions about the rotor imbalance.

4.5 Logistic Regression

4.5.1 About the algorithm

Logistic regression (LR) is a modified version of linear regression that is adapted for classification problems. Logistic regression minimizes the squared error of a linear combination of the input parameters when passed through a sigmoid function. To fully understand logistic regression, a brief background of linear regression is required.

Linear regression is the process of minimizing the least squares cost of a data set with

a linear function. This is a commonly utilized method in curve fitting, but can also be applied to very high order systems. Mathematically, this means minimizing the cost function shown in Equation 4.4. The hypothesis is the linear curve fit shown in Equation 4.5.

$$J\left(\vec{\theta}\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2 \tag{4.4}$$

$$h_\theta(x) = \mathbf{x}\cdot\vec{\theta}^T \tag{4.5}$$

$$\vec{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3, \ldots, \theta_n] \tag{4.6}$$

n is the number of parameters (or features) in the input, $\mathbf{x}$. m is the number of training examples, which means $\mathbf{x}$ is of size $[m, n]$ and $\vec{\theta}$ is a vector with n elements.

In order to apply linear regression to classification problems, the sigmoid function is applied to the linear regression hypothesis (h_θ). The sigmoid function is defined in Equation 4.7 and is visually shown in Figure 4.6. The sigmoid function is designed to map an input to a value between 0 and 1, which represents the probability that the input is of a certain class. This sigmoid function is what enables logistic regression to calculate binary outputs.

$$g(z) = \frac{1}{1+e^{-z}} \tag{4.7}$$

This new hypothesis function (Equation 4.9) produces a new cost function, shown in Equation 4.8.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[y_i \ln(h_\theta(x_i)) + (1-y_i)\ln\left(1-h_\theta\right)\right] \tag{4.8}$$

$$h_\theta = g(\mathbf{x}\cdot\vec{\theta}^T) = \frac{1}{1+e^{-(\mathbf{x}\cdot\vec{\theta}^T)}} \tag{4.9}$$

The goal of logistic regression is to create a model that best fits the experimental training data by optimizing the parameters $\vec{\theta}$ to minimize the cost function, $J(\theta)$. To

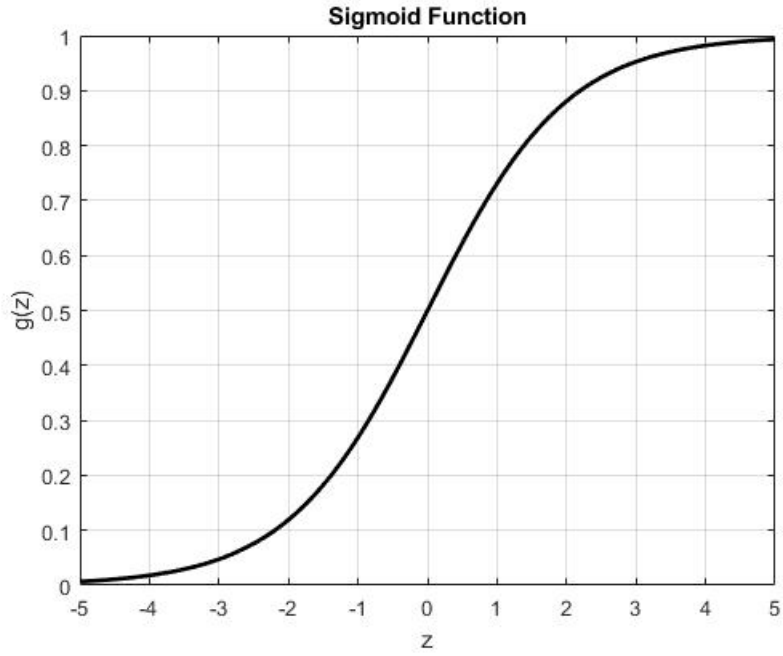

Figure 4.6: This figure shows the sigmoid function plot. This figure was created with MATLAB.

do this, the gradient descent method is used [9]. To perform gradient descent, both the cost function and the gradient function of $\vec{\theta}$ are required. The gradient can be numerically calculated using the finite difference method [14]; however, this is much more computationally intensive than directly calculating the gradient analytically. The derivative of the sigmoid function is shown in Equation 4.10. This produces a simple equation for the gradient of the cost function, which is shown in Equation 4.11.

$$g\prime(z) = g(z)(1 - g(z)) \tag{4.10}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y(i)\right) x_{j,i} \tag{4.11}$$

h_θ is defined in Equation 4.9 and $g(z)$ is defined in Equation 4.8. In other words, Equation 4.11 represents an average of $(h-y)x$ for every training example and calculated with respect to each θ parameter (j is the number of parameters in the model).

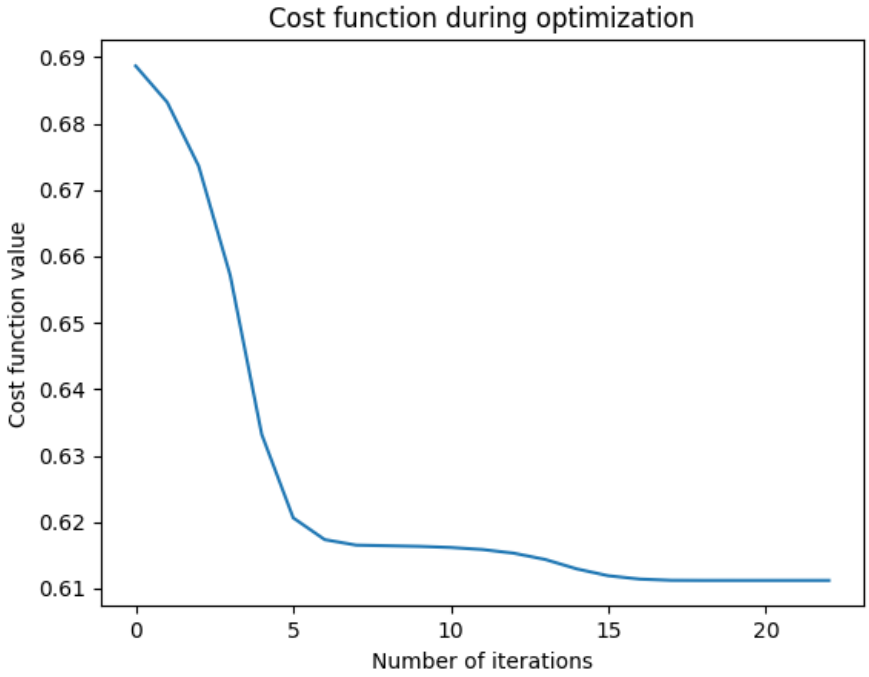

Figure 4.7: This figure shows the cost function during the training process. This model uses 2 features, so the model is trying to optimize $\vec{\theta} = [\theta_0, \theta_1, \theta_2]$ (3 parameters because there is a hidden bias unit). This figure was created with Python.

4.5.2 Training the model

To train the model, the cost function is minimized using a version of the gradient descent optimization process. This iterates through various values of θ that are calculated using the gradient of the cost function. Figure 4.7 shows the cost function over the iteration (training) process. When the cost function has converged to a minimum, the model training is complete and the model is ready to be validated on the test data.

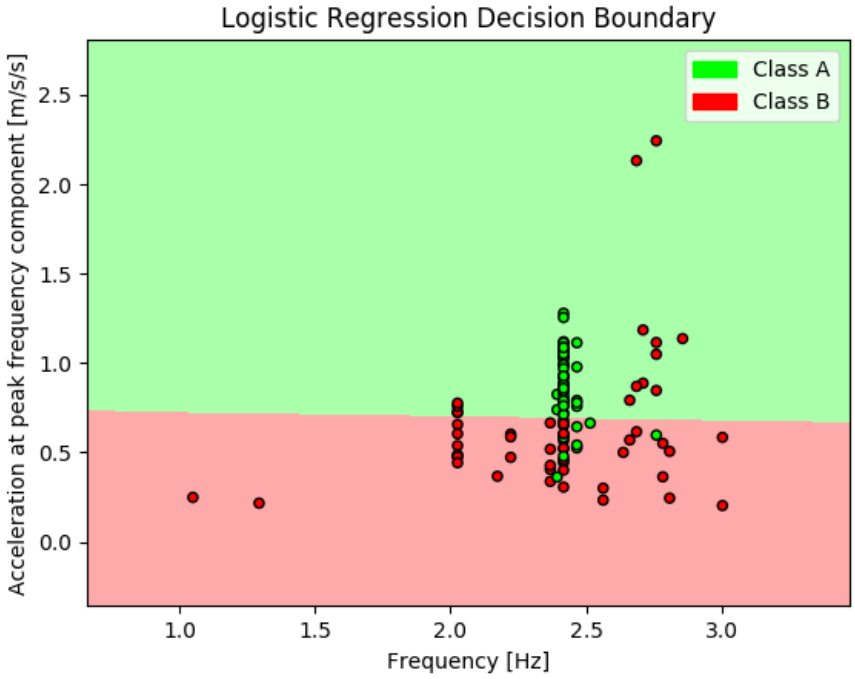

Figure 4.8: This figure shows the decision boundary from the logistic regression model. This is a linear model using 2 input parameters and a binary output class. This model has an accuracy of about 67% when using the test data set. This figure was created with Python.

4.5.3 Applying the model

Minimizing the cost function results is a trained model that produces the decision boundary shown in Figure 4.8. This model has a pretty poor accuracy of about 67% on the test data.

To improve this model, the 2 input parameters can be expanded to create a higher order models. For example, the 2 input features for a linear logistic regression model are shown in Equation 4.12. To achieve higher order models, the input parameters can be expanded to included the terms for the higher order polynomials. This will allow the regression to fit parameters to each of the new inputs. Equation 4.13 represents

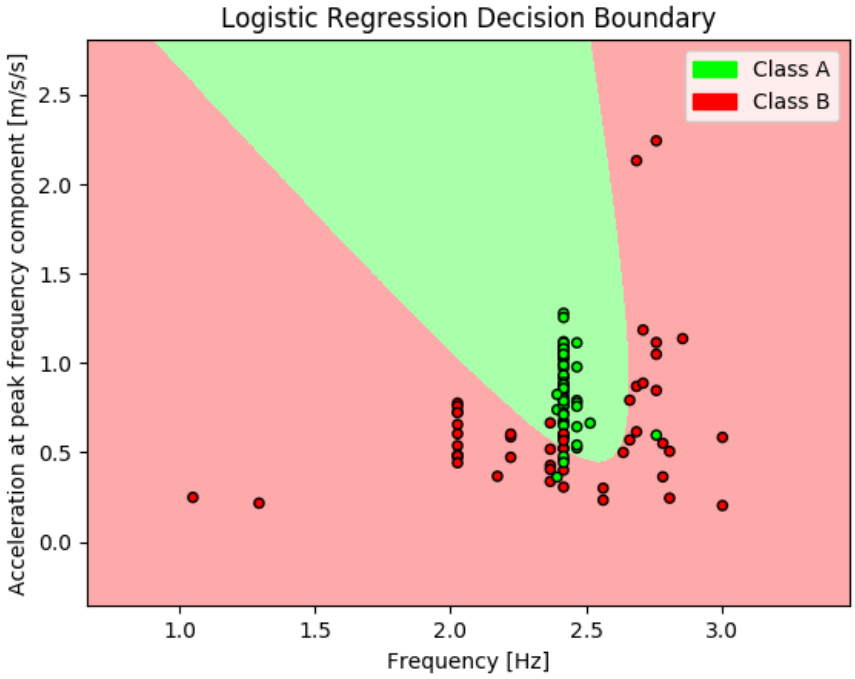

Figure 4.9: This figure shows the decision boundary from the logistic regression model. This is a 2nd order model using 2 input parameters and a binary output class. This model has an accuracy of about 85% when using the test data set. This figure was created with Python.

the input variables expanded for a 2nd order model and Equation 4.14 represents the input variables expanded for a 3rd order model. Figure 4.9 shows the decision boundary for a 2nd order model and Figure 4.10 shows the decision boundary for a 3rd order model.

$$[x_1, x_2] \tag{4.12}$$

$$[x_1, x_2, x_1^2, x_1x_2, x_2^2] \tag{4.13}$$

$$[x_1, x_2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^2] \tag{4.14}$$

In fact, the order can keep getting increased until the model fits the training data perfectly. This is called over-fitting and is a problem with higher order machine learning models. Figure 4.11 shows an example of a 10th order model that has some over-

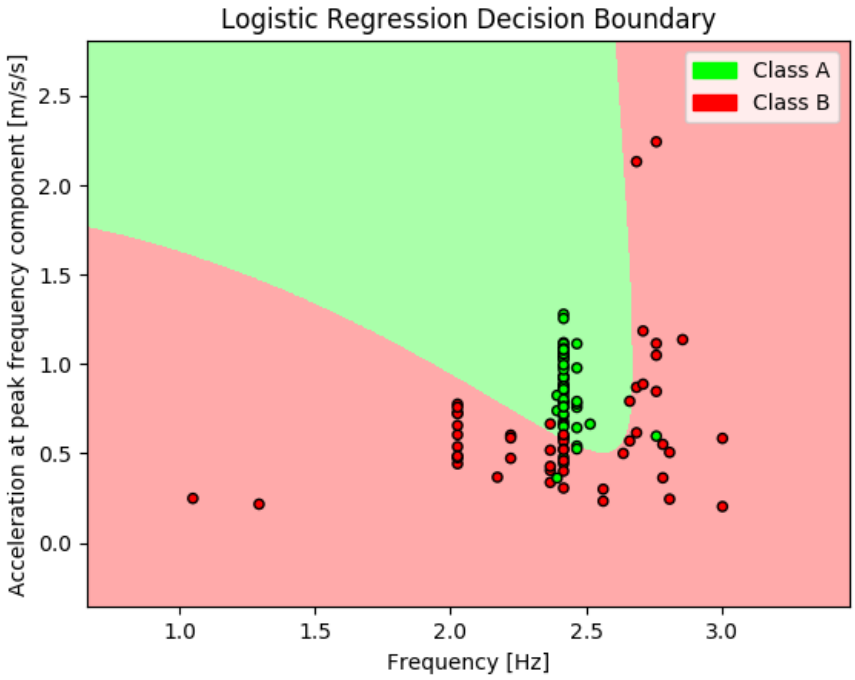

Figure 4.10: This figure shows the decision boundary from the logistic regression model. This is a 3rd order model using 2 input parameters and a binary output class. This model has an accuracy of about 85% when using the test data set. This figure was created with Python.

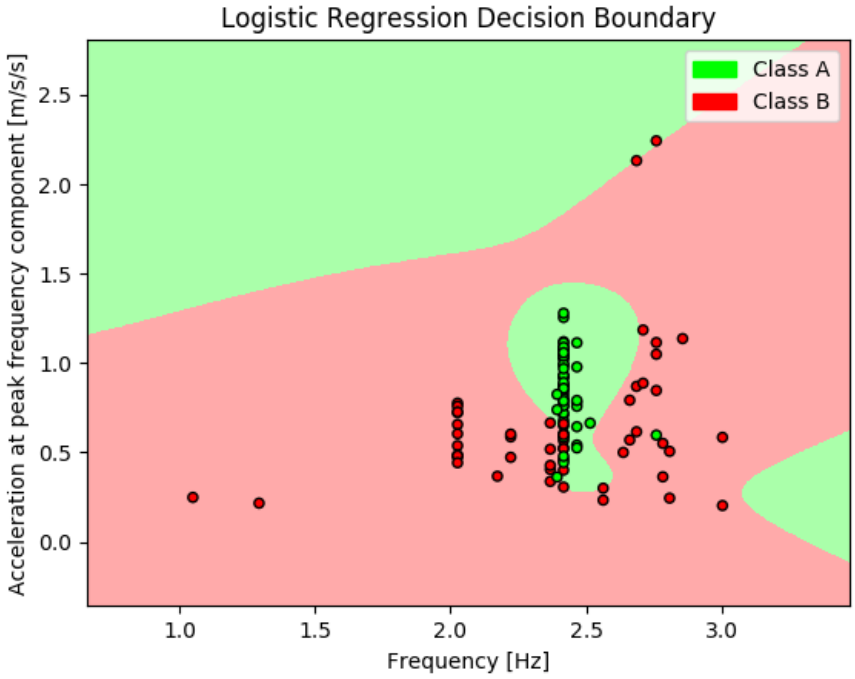

Figure 4.11: This figure shows the decision boundary from the logistic regression model. This is a 10th order model using 2 input parameters and a binary output class, and has an over-fitting problem. This model has an accuracy of about 81% when using the test data set. Notice that the accuracy of the over-fitting model on the test data is lower than the lower order models shown in Figure 4.9. This figure was created with Python.

fitting problems. It is unlikely that the small region separating the Class A sections (at about 2.5 Hz and 1.5 m/s/s) belongs to Class B. In order to solve the problem of over-fitting, a regularization term can be added to the cost function equation. This regularization term, λ, effectively reduces the impact that the optimization parameters have on the cost calculation, which keeps the optimized parameters small. This reduces the "strength" of the regression model and can help eliminate over-fitting problems. Equation 4.15 shows the new logistic regression cost function with a regularization term applied. The new gradient function can be derived with the regularization term, as shown in Equation 4.16.

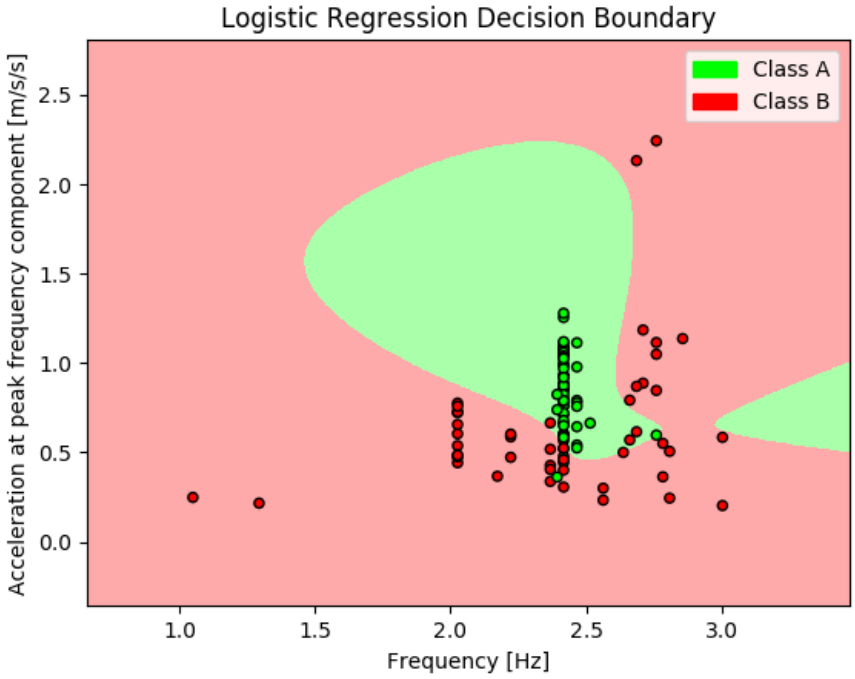

Figure 4.12: This figure shows the decision boundary from the logistic regression model using a regularization term of $\lambda = 3$. This is a 10th order model using 2 input parameters and a binary output class. This model has an accuracy of about 86% when using the test data set, which is higher than the 10th order model with no regularization (Figure 4.11). This figure was created with Python.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[y_i \ln(h_\theta(x_i)) + (1-y_i)\ln\left(1-h_\theta\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{4.15}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y(i)\right)x_{j,i} + \frac{\lambda}{m}\theta_j \tag{4.16}$$

Adding regularization to the 10th order model shown in Figure 4.11, can result in a much better fitting model for the data. Figure 4.12 shows the result of a regularization term of $\lambda = 3$.

The logistic regression function outputs the probability that the data set falls into a certain class. This means that for most decision boundary plots, the data set is

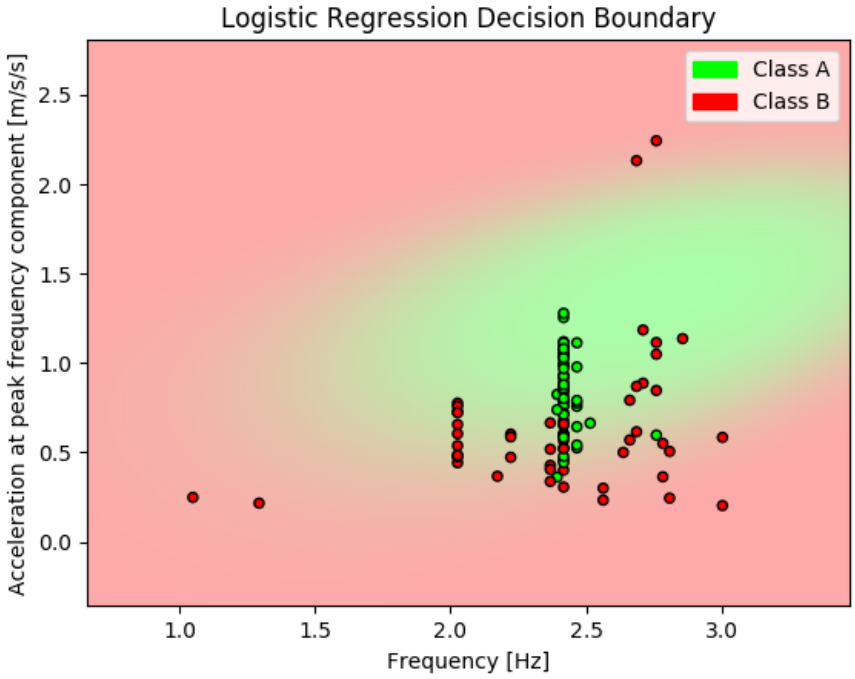

Figure 4.13: This figure shows the probability of the model classifying certain data sets as Class A and Class B. This model was trained with an order of 3 and a regularization parameter of $\lambda = 1$. The background color of the plot represents the probability of the data set being classified as Class A, with solid green being about 99% certainty and solid red being about 1% certainty. This figure was created with Python.

classified as Class A if the probability is greater than 50% and the data is classified as Class B if the probability is less than 50%. The decision boundary can also include the probability to produce a smooth gradient plot as in Figure 4.13.

4.5.4 Using a complete feature set

The logistic regression models in the previous section use only 2 features from the entire data set (the features shown in Figure 4.4). This is convenient because it is easy to conceptualize and visualize. The decision boundaries for models with 2 features can be analyzed using plots like Figure 4.12. A better version of the model would use

all of the frequency spectrum data as inputs to the model. This means that instead of using the peak amplitude and corresponding maximum frequency as model inputs, the entire FFT could be used.

Using the whole frequency spectrum means the model will have 256 features (one for the magnitude of each frequency component). This is a lot of features to use for the KNN algorithm and will be far too slow when more training data is acquired. The benefit of regression-based models is that the computational cost to predict the class of new data remains the same regardless of how many training examples are used to optimize the model parameters. This means that using more data will just improve the model without slowing down the real-time calculations (it will, however, make the training time much slower).

Figure 4.14 shows a plot of the cost function value during the training process. This is a common way of visually ensuring the model has converged to a local minimum. This produces an accuracy score of 95% on the test data which is much higher than the scores from the 2-feature models in the previous section.

To better understand the results of the training process, the model parameters are plotted in Figure 4.15. This shows each of the parameters in $\vec{\theta}$ and the corresponding frequencies they are correlated to. To calculate the probability of the turbine being in a specific class, these parameters are multiplied with the FFT results and passed through the sigmoid function, as shown in Equation 4.9.

From inspecting Figure 4.15, it appears as though the frequency around 2.5 Hz is weighted the most, with some of the surrounding frequencies having small negative weights. The logistic regression model essentially "learns" that it should inspect the rotor frequency to classify the state of the turbine. It is also interesting that frequencies +/- 0.5 Hz from the strongest component in $\vec{\theta}$ have the biggest negative

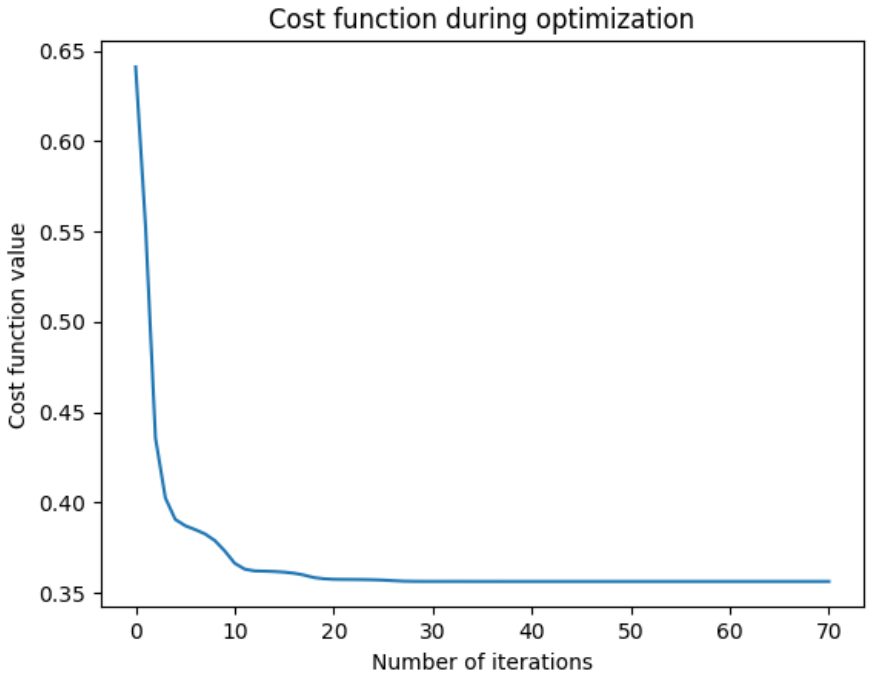

Figure 4.14: This figure shows the cost function value during the training process with a regularization term of $\lambda = 1$. In this model, all 256 features are used as inputs. This figure was created with Python.

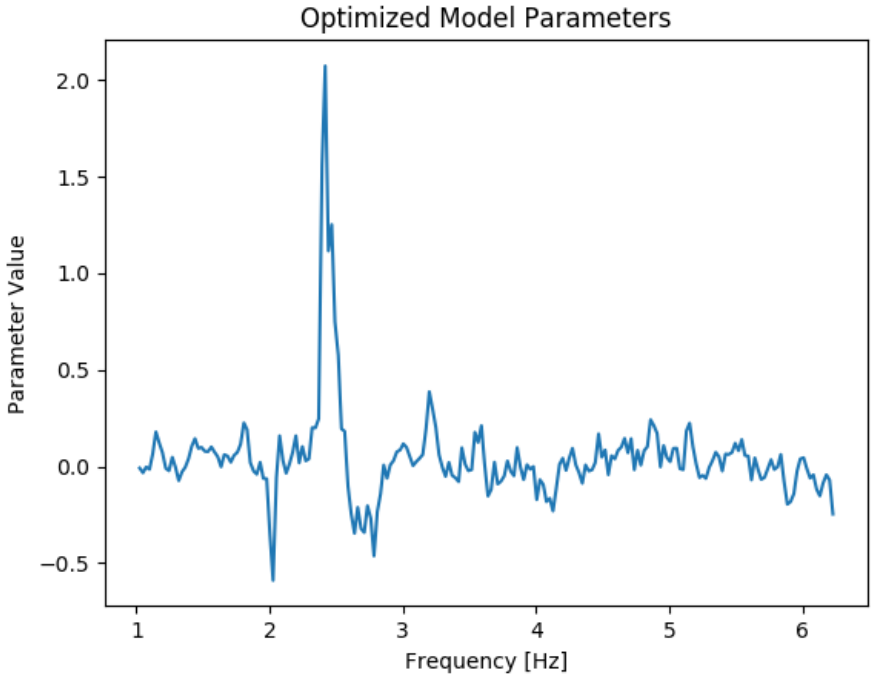

Figure 4.15: This figure shows the optimized model parameters for the logistic regression model using all the input features. Each parameter corresponds to a specific frequency, which means that this function gets multiplied by the FFT of each data set and used to calculate the probability that the turbine is either balanced on not balanced. This figure was created with Python.

weights. From a visual inspection of the training data in one of the previous decision boundary plots (like Figure 4.12), it appears as though data is likely to belong to Class A if the frequency is fairly constant around 2.5 Hz. If the frequency diverges from this value, the training data generally falls into Class B, which is how the logistic regression model is analyzing the data and predicting the turbine state.

4.6 Neural Network

4.6.1 About the algorithm

A neural network is an algorithm that resembles the neurons in the brain. This is essentially a more complicated version of logistic regression, and is capable of training nonlinear models. In some of the previous models (Such as the models in Figure 4.5 and Figure 4.8), the DFT data was condensed down into 2 variables, which are the magnitude of the maximum frequency component and its corresponding frequency. These variables were selected from intuition about the tower dynamics; however, there may be 2 different abstract features that better describe the system. In order to find the 2 best features that describe the system, a neural network can be set up as shown in Figure 4.16. This network has 3 layers, which include the input layer, hidden layer, and output layer. The hidden layer contains 2 units, which are abstract features that can be fit to a linear model. These 2 features should be a better indicator of the state of the turbine than the frequency and magnitude values previously used. The (+1) units in the network represent bias units that are injected into the network to apply a constant bias to each set of parameters or features.

For each layer, the value of each unit can be calculated by multiplying the weight matrix by the previous layer units and applying the sigmoid function, $g(z)$ (Equation 4.7). For example, the values in layer 2 ($\mathbf{a_2}$) can be calculated using Equation 4.17.

$$\mathbf{a_2} = g\left(\mathbf{X} \cdot \mathbf{\Theta_1}\right) \tag{4.17}$$

In Equation 4.17, $\mathbf{X}$ is the input data of size $[m, n]$, where m is the number of training examples, and n is the number of features (256 if all the DFT values are used). $\mathbf{\Theta_1}$ is the matrix of weight values that is optimized during the training process.

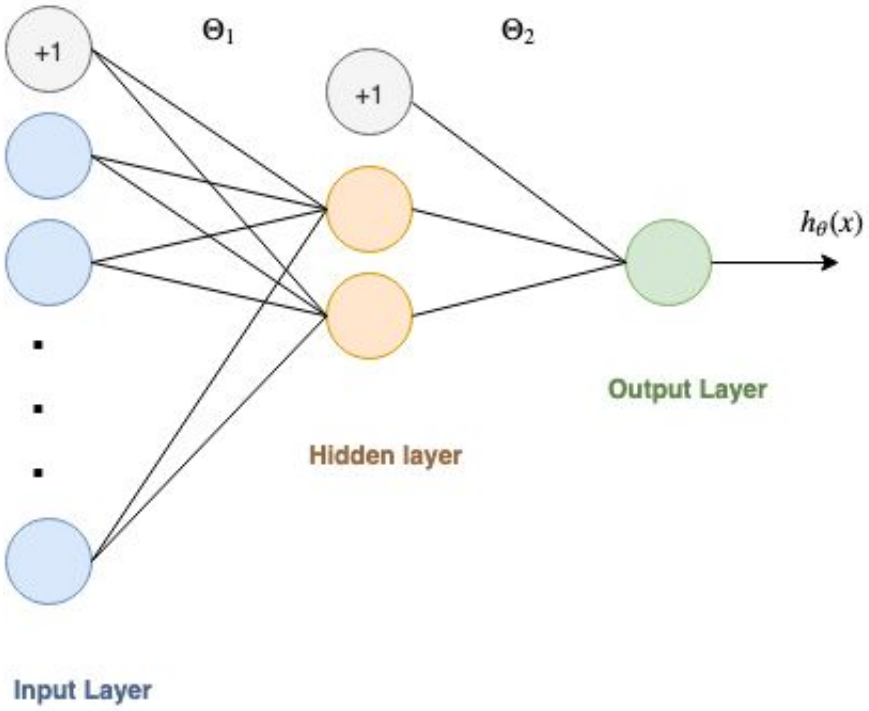

Figure 4.16: This figure shows a neural network diagram for the turbine model. The input layer contains the 256 DFT values, and the output layer contains the predicted probability that the turbine is balanced or not balanced. The hidden layer represents some abstract features that can be used to predict the state of the turbine with a linear model. This figure was created with Draw.io.

The hypothesis of the model in Figure 4.16 is calculated using Equation 4.18. This is the predicted probability that the turbine is balanced or unbalanced for all the training examples.

$$\vec{h}_\theta = g\left(\mathbf{a_2} \cdot \mathbf{\Theta_2}\right) \tag{4.18}$$

The squared difference cost function of the neural network diagram is shown in Equation 4.19.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_{k,i}\ln\left(h_\theta(x_i)\right) - (1-y_{k,i})\ln\left(1-h_\theta(x_i)\right)\right] \tag{4.19}$$

To calculate the analytic gradient of the cost function back propagation is required. Back propagation is an algorithm is derived from calculus to determine the gradient of a neural network. As the name suggests, this method starts from the neural network output and calculates the error of each layer, which can be related to the gradient. First, δ for each layer is calculated as shown in Equation 4.20 and Equation 4.21. $\odot$ is the element-wise operator, which can also be represented as $.*$ in MATLAB. It is also important to remove the term for $\delta_{2,0}$ because there is no gradient term defined for the bias unit in the second layer.

$$\vec{\delta}_3 = \vec{h} - y \tag{4.20}$$

$$\vec{\delta}_2 = \mathbf{\Theta_2}^T \cdot \vec{\delta}_3 \odot g\prime\left(\mathbf{X\Theta_1}\right) \tag{4.21}$$

Using δ, the gradient of $\mathbf{\Theta_1}$ and $\mathbf{\Theta_2}$ can be calculated using Equation 4.22 and Equation 4.23.

$$\frac{\partial J(\Theta)}{\partial \Theta_1} = \frac{1}{m}\left(\vec{\delta}_2^T \cdot \vec{a}_1\right) \tag{4.22}$$

$$\frac{\partial J(\Theta)}{\partial \Theta_2} = \frac{1}{m}\left(\vec{\delta}_3^T \cdot \vec{a}_2\right) \tag{4.23}$$

The back propagation method for calculating the gradient is much faster than using a numerical method such as the finite difference approximation. Many optimization

algorithms will default to the forward finite difference gradient approximation if a gradient function is not supplied, so it is important to provide the optimization function with both the cost function and gradient function.

4.6.2 Training the model

Now that a neural network model has been set up and mathematically described, the model needs to be trained. In other words, this means finding values for $\mathbf{\Theta_1}$ and $\mathbf{\Theta_2}$ that minimize the cost function, $J(\Theta)$. Optimization functions generally assume the parameters are in a single vector (rather than 2 separate matrices). $\mathbf{\Theta_1}$ and $\mathbf{\Theta_2}$ can be flattened into a single vector, and unrolled into matrices when they are used in calculations. An example of flattening the parameters into a single vector using Python (with the numpy package) is shown below:

```
nn_params = numpy.append(Theta1.flatten(), Theta2.flatten())
```

To train the model, the parameters are minimized using the `fmincg` function from the `scipy.optimize` package. This is a nonlinear conjugate gradient algorithm that was developed by Polak and Ribliere [26]. Figure 4.17 shows the cost function during the optimization process. *It is very important that the parameters* $\mathbf{\Theta_1}$ *and* $\mathbf{\Theta_2}$ *are initialized randomly before the training process.* If all of the values are initialized to zero, the model will not develop distinct weights for each connection. This is a property of neural networks and is usually standard practice when training models.

4.6.3 Applying the model

Minimizing the cost function results in optimized parameters for $\mathbf{\Theta_1}$ and $\mathbf{\Theta_2}$. Figure 4.18 shows a decision boundary plot using the hidden layer activations as input vari-

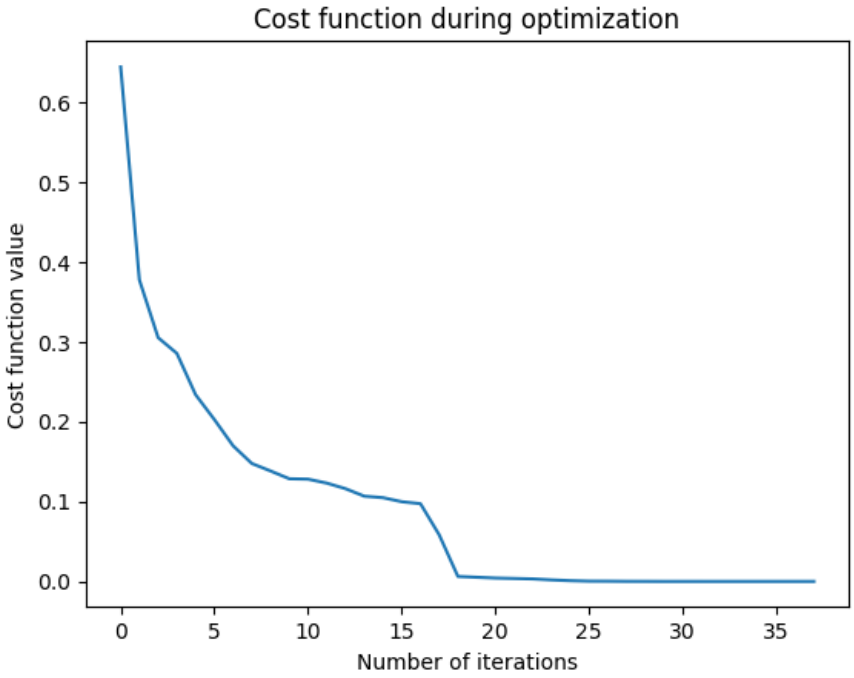

Figure 4.17: This figure shows the cost function of the neural network in Figure 4.16 during the optimization process. This figure was created with Python.

ables. When this plot is compared to the 2-variable data in Figure 4.12, the variables in Figure 4.18 seem to be a much better choice of inputs. These are abstract variables that can be fit to a linear model (notice the straight line defining the decision boundary) to describe the state of the turbine.

For clarification, the data shown in Figure 4.18 are internal variables used in the neural network classification algorithm and are created from the same 256-point DFT data as the previous plots. This model has a prediction accuracy of 85 - 95% on the test data depending on how the variables are randomly initialized. This variability is a result of a lack of training and test data. For a proper neural network model more data is always better. Unlike the nonlinear KNN algorithm, neural networks are nonlinear algorithms that have a relatively constant prediction time. More data will not change the computational time required to calculate an output with a new test data set.

4.6.4 Deep Learning

A deep learning network is a neural network with multiple layer of nonlinear processing units. The previous model was designed using the structure in Figure 4.16, which was chosen because the hidden layer only has 2 units. This makes it easy to visualize the intermediate results of the model. More complicated neural network models are possible, but they are harder to conceptualize and visualize. For example, Figure 4.19 shows a version of a 6-layer neural network with 4 hidden layers. The multiple layers in deep neural networks are abstract representations of system features.

To design a deep neural network, the Keras [20] API for TensorFlow [16] is used to simplify the process. This model has 4 hidden layers, each with 50 units per layer, which is a total of 18,551 parameters that need to be optimized when the 256 FFT

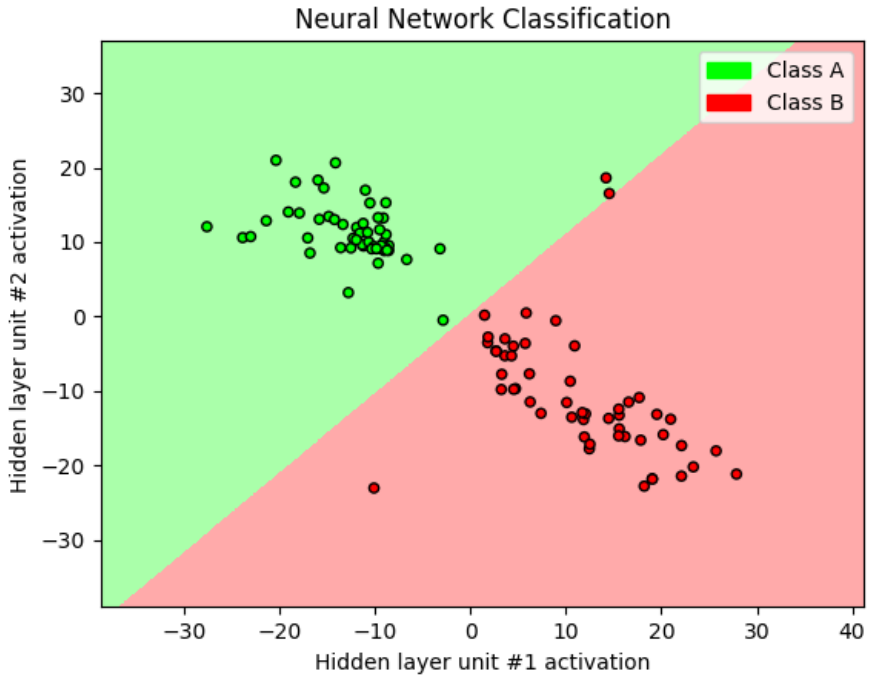

Figure 4.18: This figure shows the decision boundary of the hidden layer activations of the neural network model shown in Figure 4.16. These 2 parameters are the abstract variables that the network created to linearly classify the system. This figure was created with Python.

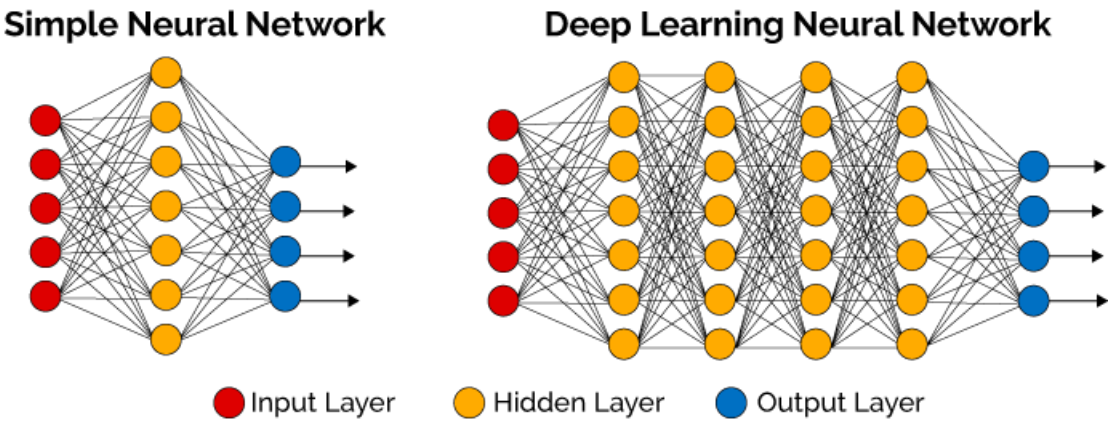

Figure 4.19: This figure shows an example of a deep learning network compared with a simple neural network (like the one in Figure 4.16) [31].

Table 4.2: Deep neural network design

Layer	Number of Units	Activation Function
Input Layer	256	None
Hidden Layer 1	50	ReLU
Hidden Layer 2	50	ReLU
Hidden Layer 3	50	ReLU
Hidden Layer 4	50	ReLU
Output Layer	1	Sigmoid

values are used as inputs (shown in Table 4.2). The ReLU activation function is the rectified linear unit, which is defined as g_{relu} (Equation 4.24). ReLU functions are commonly used for deep neural network hidden layer activation functions because they are cheap to compute and generally converge faster. The sigmoid function is still required for the output layer because this needs to calculate a probability.

$$g_{relu}(x) = \max(0, x) \tag{4.24}$$

This is a complex model that cannot be visualized using a decision boundary plot like

the previous models. Complex models can be prone to overfitting, which means they effectively memorize the training data and perform poorly on new test data. Epochs, or iterations, can be manually set to limit model overfitting. For every epoch, the training loss should be decreasing because the optimization process is taking a new step towards minimizing the loss. If the model is actually improving, the test data loss should also be decreasing. Once the model begins to start overfitting, the test loss will start to increase. This shows that the model has started to memorize the training data and is not doing a good job at predicting outputs on the new (test) data. Figure 4.20 shows a plot of the training and test data loss over each epoch. It can be seen that the optimum amount of epochs is about 200, because more will cause the testing loss to increase. This can be verified in Figure 4.21, which shows the test and training data reach a maximum accuracy at about 200 epochs also. Any more iterations (epochs) will cause the model to overfit the training data and will not improve the calculated accuracy.

When the iteration process is stopped after 200 epochs, the training data accuracy is 97%; however, this is not expected to be a good representation of the performance of the neural network. Test data accuracy is a much better representation because it tests the model on new data. The test data accuracy for this model is about 91%.

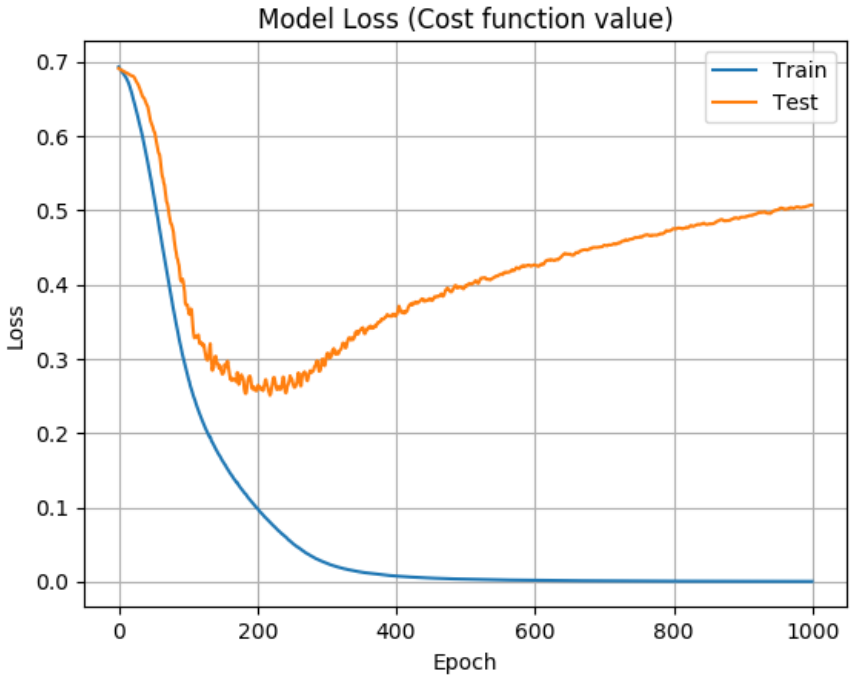

Figure 4.20: This figure shows the loss (cost function value) for the training and test data using the model with a structure shown in Table 4.2. This figure was created with Python.

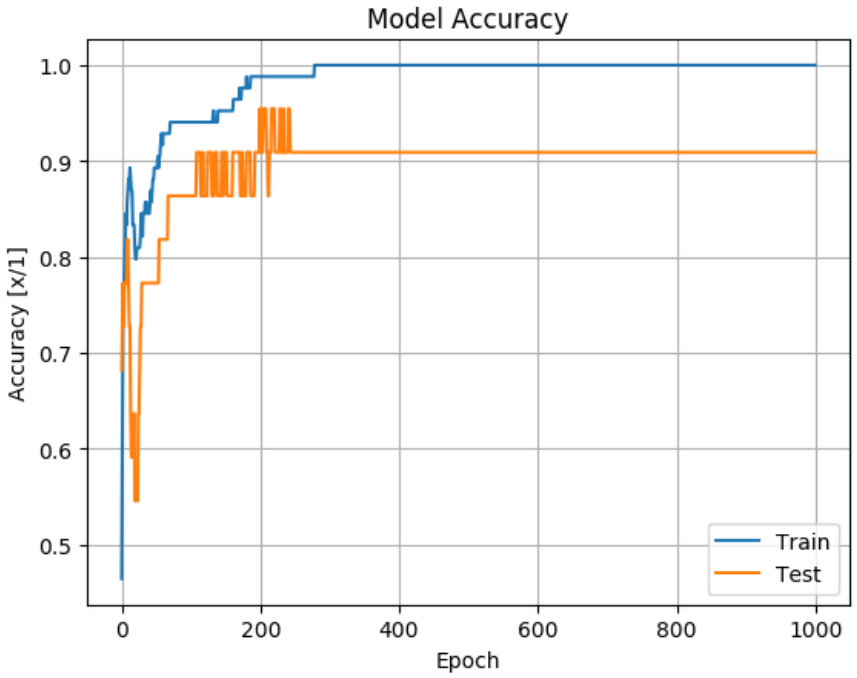

Figure 4.21: This figure shows the accuracy of the training and test data using the model with a structure shown in Table 4.2. This figure was created with Python.

Chapter 5

CONCLUSION

5.1 Summary

There are many ways to process the experimental acceleration data from the tower, depending on the computational power and stability of the rotor speed. A lock-in amplifier and the Goertzel algorithm are good methods for calculating a single frequency component. These work well when the signal of interest falls at a known frequency, such as if the rotor speed is being measured independently. Additionally, Zoom FFTs can be used to significantly reduce the computational load if only a small frequency region is of interest. Zoom FFTs involve shifting, filtering and decimating the raw data before it is translated into the frequency domain using a standard FFT algorithm. This method is ideal if the target frequency range is much smaller than the sampling rate of the system. For example, a Zoom FFT is a good choice if the acceleration data is sampled at 128 Hz, but the rotor frequency only varies from 2-4 Hz. If the classification algorithm uses the entire frequency spectrum as feature inputs, the performance can be significantly improved by removing the frequency components that don't contain useful information, such as any frequency significantly higher than the rotor frequency.

The classification algorithms are a type of supervised machine learning, which means they use training data that has pre-defined labels. Each experimental data set must be be labeled as either 'good" or "bad" (balanced/not balanced) prior to training the model. The K-Nearest Neighbors (KNN) algorithm works well with small data sets because it just stores all the training data in memory and uses this information to

determine how close a new data set is to previous data sets. The KNN algorithm does not scale well, so it is not a good choice when more data is obtained of if more features (input parameters) are used. Standard logistic regression is the process of fitting a linear model to data with discrete class outputs. This model does not work well with just 2 features (maximum magnitude and corresponding frequency), because they cannot be separated by a linear function. Logistic regression can be modified to include higher order terms to better fit the experimental data, or more features can be used.

If all 256 FFT points are used as input features, logistic regression performs very well and is easy to conceptualize because it is still a linear model. This type of model benefits from more experimental data to better train and test the model with different data sets. To take the classification algorithm even further, abstract features can be added to the logistic regression model to create neural networks. When many abstract feature layers (hidden layers) are added, these models can be classified as deep neural networks. Neural networks are great for modeling complex non-linear systems, or when the amount of input features is large. A neural network is a great classification algorithm structure to have that will allow easy scalabilty when more data is collected or more features are measured.

5.2 Next Steps

The immediate next step is to collect more data. Machine learning algorithms require lots of training data to optimize the internal parameters. A good reference is to have about 10 times the amount of training data as features or trainable parameters. This project used about 100 data sets for 256 features, so ideally, there would be at least 2560 data sets to train the models.

The turbine tower model can be expanded to include a more complicated forced input or a more complicated mechanical model. If the model is accurate enough, it can be used to simulate balances and imbalances and create training data for the machine learning algorithm. Additionally, this would allow the machine learning algorithm to be trained on datasets that represent turbine failures and could be too dangerous or costly to experimentally collect.

In addition to collecting more raw data, it would be best to collect data over a variety of operating conditions. For example, running the turbine for 20 hours and collecting consecutive data won't account for day-to-day fluctuations in wind speed, weather, and temperature. These new data sets might show the need to new features to be added, such temperature, wind speed, or acceleration direction as well as magnitude.

After achieving a successful classification model, it might be desirable to reduce the power usage of the detection device. This would most likely involve switching to a low power processor and reducing the amount of calculations that can be performed. This would require optimized versions of frequency transformations, such as Zoom FFTs or lock-in amplifiers.

Finally, a deeper investigation into machine learning classification algorithms can be done to choose the ideal model for this application. For example, it might be useful to use an unsupervised algorithm that can create its own groups for the data, or maybe a neural network with a more complicated architecture can be used to better fit the data.

BIBLIOGRAPHY

[1] K-nearest neighbors. *Scholarpedia*, 2009.

[2] About lock-in amplifiers. *thinkSRS*, 2017.

[3] Envelope detection. *MathWorks*, 2018.

[4] Draw.io. 2019.

[5] S. Asian, G. Ertek, C. Haksoz, S. Pakter, and S. Ulun. Wind turbine accidents: A data mining study. *IEEE Systems Journal*, 11:1567–1578, 09 2017.

[6] L. Atkinson. A simple benchmark of various math operations. 2014.

[7] Atul. K-nearest neighbors algorithm using python. 2019.

[8] K. Banks. The goertzel algorithm. *Embedded*, 2002.

[9] M. Cheatsheet. Gradient descent. 2017.

[10] D. A. R. Collins. Zoom fft.

[11] DataSchool. Simple guide to confusion matrix terminology. 2014.

[12] R. M. Edward Malnick. 15000 accidents and incidents on uk wind farms. *Telegraph*, 2011.

[13] B. Evans. Goertzel algorithm. *Berkeley*.

[14] B. Fornberg. Finite difference method. *Scholarpedia*, 6(10):9685, 2011. revision #121475.

[15] R. George et al. Transient small wind turbine tower structural analysis with coupled rotor dynamic interaction. 2013.

[16] Google. Tensorflow. 2019.

[17] T.-g. Gwon. Structural analyses of wind turbine tower for 3 kw horizontal-axis wind turbine. *Cal Poly Masters thesis*, 2011.

[18] T. Instruments. Modified goertzel algorithm in dtmf detection using the tms320c80. *Digital Signal Processing Solutions*, 1996.

[19] I. International. Decimation. *dspGuru*, 2017.

[20] Keras. Keras. 2019.

[21] R. Lyons. Using zoom fft for spectral analysis. *Embedded*, 2011.

[22] E. G. Maik Reder, Julio Melero. Wind turbine failures - tackling current problems in failure data analysis. *Journal of Physics Conference Series*, 2016.

[23] MathWorks. Blackman window. 2019.

[24] W. J. McCutcheon. Deflections and stresses in circular tapered beams and poles. *Civ. Eng. Pract. Des. Eng*, 2, 1983.

[25] M. L. Meade. *Lock-in amplifiers: principles and applications*. Number 1. Mike Meade, 1983.

[26] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006.

[27] H. J. Nussbaumer. *Fast Fourier transform and convolution algorithms*, volume 2. Springer Science & Business Media, 2012.

[28] D. I. Peng Guo. Wind turbine tower vibration modeling and monitoring by the nonlinear state estimation technique (nset). *MDPI*, 2012.

[29] W. H. Press and S. A. Teukolsky. Savitzky-golay smoothing filters. *American Institute of Physics*, 1990.

[30] M. Püschel and J. M. Moura. Algebraic signal processing theory: Cooley-tukey type algorithms for dcts and dsts. *arXiv preprint cs/0702025*, 2007.

[31] Rajat. Mnist vs mnist. 2018.

[32] J. O. Smith. *Introduction to digital filters: with audio applications*, volume 2. Julius Smith, 2007.

[33] ST. Floating point unit demonstration on stm32 microcontrollers. 2016.

[34] A. D. B. Thomas Kenbeek, Stella Kapodistria. Data driven online monitoring of wind turbines. 2016.

[35] Wikipedia contributors. Finite impulse response — Wikipedia, the free encyclopedia, 2018.

[36] Wikipedia contributors. Infinite impulse response — Wikipedia, the free encyclopedia, 2018.

[37] S. Winograd. On computing the discrete fourier transform. *Proceedings of the National Academy of Sciences*, 73(4):1005–1006, 1976.

APPENDICES

Appendix A

SOURCE CODE

The source code for this project can be found at my GitHub: `https://github.com/ryan-takatsuka/masters_thesis-wind_turbine`