2017-12-01

Eigenblades: Application of Computer Vision and Machine Learning for Mode Shape Identification

Alex W. La
Brigham Young University

Eigenblades: Application of Computer Vision and Machine Learning for Mode Shape Identification

Alex W La

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

John L. Salmon, Chair
Steven Gorrell
Bryan Morse

Department of Mechanical Engineering

Brigham Young University

ABSTRACT

Eigenblades: Application of Computer Vision and Machine Learning for Mode Shape Identification

Alex W La
Department of Mechanical Engineering, BYU
Master of Science

On August 27, 2016, Southwest Airlines flight 3472 from New Orleans to Orlando had to perform an emergency landing when a fan blade separated from the engine hub and destroyed the cowling and punctured the fuselage. Initial findings from the metallurgical examination conducted in the National Transpiration Safety Board Materials Laboratory found that the fracture surface of the missing blade showed curving crack arrest lines consistent with fatigue crack growth. Fatigue is often cause by resonate vibrations. Modal analysis is a method that can model the natural frequencies and bending modes of turbomachinery blades. When performing modal analysis with finite element solvers like Mechanical ANSYS, images are often generated to help an engineer identify mode shapes created by nodal displacements. Manually identifying mode shapes from these generated images is an expensive task. This research proposes an automated process to identify mode shapes from gray-scale images of turbomachinery blades within a jet-engine. This work introduces mode shape identification using principal component analysis (PCA), similar to approaches in facial and other recognition tasks in computer vision. This technique calculates the projected values of potentially linearly correlated values onto P-linearly orthogonal axes, where P is the number of principal axes that define a subset space. Classification was performed using support vector machines (SVM). Using the PCA and SVM algorithm, approximately 5300 training images, representative of 16 different modes, were used to create a classifier. A test set was created with approximately 2000 unknown mode images. The classifier achieved on average 98.4% accuracy on the test set when using the bilinear Eigenface method. The accuracy was 98.6% when using the triangle interpolate Eigenface method. In addition, The results suggest that using digital images to perform mode shape identification can be achieved with better accuracy and computation performance compared to previous work. Potential generalization of this method could be applied to other engineering design and analysis applications.

Keywords: Modal Analysis, Computer Vision, PCA, Machine Learning

ACKNOWLEDGMENTS

I would like to acknowledge the time and effort that my advisor, Dr. John Salmon, has given to assist in this research and subsequent thesis. In addition, Dr. Steven Gorrell and Dr. Bryan Morse have been great sources of information and inspiration. Thank you to Brigham Young University for the great years. Thank you to Evan Selin, Ammon Hepworth, and Clint Collins at Pratt and Whitney for their suggestions and support. Finally, I thank my wife, Alysa, for her never-ending love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

CV	Computer Vision
ML	Machine Learning
PCA	Principal Component Analysis
PCs	Principal Components
SVM	Support Vector Machine
DOE	Design of Experiments
NURBS	Non-uniform Rational B-spline
kNN	*k*-Nearest Neighbors
RBF	Radial Basis Function
TP	True Positive
TN	True Negative
FN	False Negative
FP	False Positive
MAC	Modal Assurance Criterion

CHAPTER 1. INTRODUCTION

1.1 Problem Overview

Today, most airplanes are powered by jet engines: complex dynamic structures that spin at high velocities [2]. Analyzing the resonant vibrations of airfoil blades is an essential part of the development of compressor blades for gas turbine engines. Compressor blades are rotating structural elements that add mechanical energy to the air within jet engines. Resonant vibrations are possible when rotor blades are subjected to non-uniform pressure pulses from upstream and downstream obstructions in the flowpath. The resulting resonant vibratory stresses can produce high cycle fatigue (HCF) [3, 4] and premature failure of the airfoil [5]. A catastrophic fatigue failure of a fan blade occurred recently that forced a Boeing 737-700 to make a emergency landing. On August 27, 2016, Southwest Airlines flight 3472 from New Orleans to Orlando had to perform an emergency landing when a fan blade separated from the engine hub and destroyed the cowling and diffuser and punctured the fuselage. The damage is shown in Figure 1.1 which was taken just after the failure. Thankfully, the pilots were able to make a safe emergency landing at Pensacola where there were no injuries to the passengers or crew members. As part of an ongoing investigation by the National Transpiration Safety Board (NTSB), initial findings from the metallurgical examination conducted in the NTSB Materials Laboratory include: the fracture surface of the missing blade showed curving crack arrest lines consistent with fatigue crack growth, the center of the fatigue origin area was about 2.1 inches aft of the forward face of the blade root, and no surface or material anomalies were noted during an examination of the fatigue crack origin [6]. This example demonstrates the need to model fatigue and therefore resonate vibrations on turbomachinery blades.

Dynamic modeling is an important part of the design process and resonate vibratory stresses can be predicted by it [7]. Examining a blade's vibration mode shapes is a common approach to predicting the dynamic response of a blade [8]. A mode shape is a specific pattern of vibration at a given natural frequency. Finite element analysis (FEA) is often used to model a structure's

Figure 1.1: Catastrophic engine failure during Southwest flight 3472. Courtesy USAToday [1].

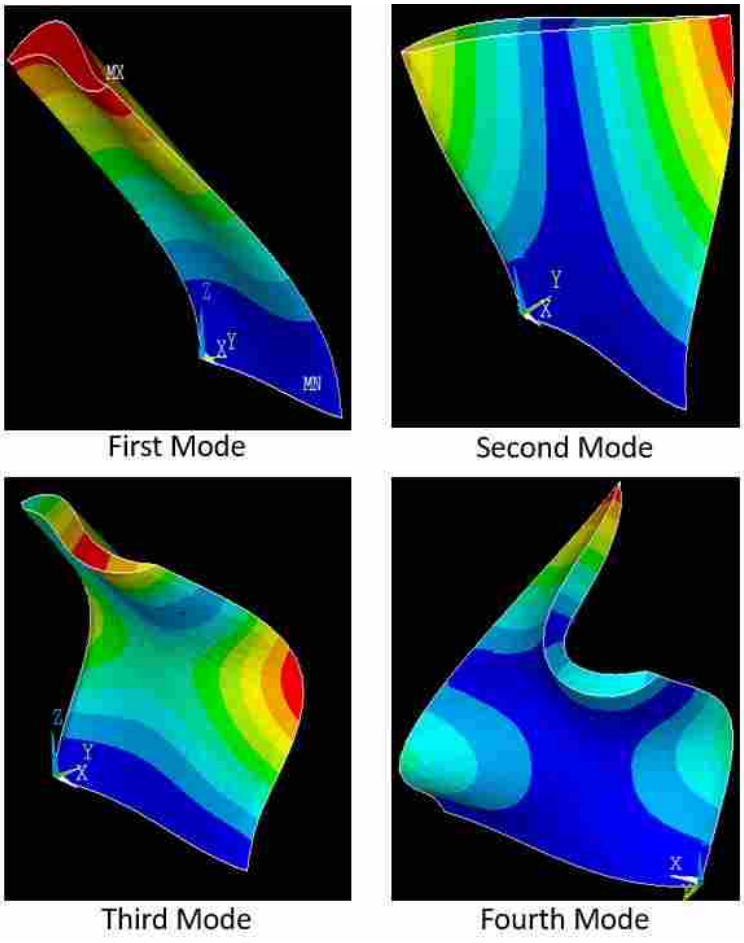

Figure 1.2: Modal Analysis - Modes at Different Natural Frequencies: 3D view. These are exaggerated displacement values of the blade shape, but these shapes occur in real physical systems. The color blue represents small displacements and red indicates a large displacement from the nominal blade.

dynamic properties. FEA solvers like Mechanical ANSYS and Siemens NASTRAN have methods that perform modal analysis. These methods allow engineers to numerically calculate the natural frequencies and mode shapes caused by structural displacement of a compressor blade. Experiments are often performed to validate these models [9]. Figure 1.2 shows the 3-dimensional (3D) modes of a generic turbomachinery blade in first bending, first torsion, chord-wise bending, and second torsion. The displacements are magnified to show how blades deflect. Figure 1.3 shows a set of 2-dimensional (2D) modes. These contour images are the 2D projections of the displacement at a natural frequency that occur in 3D.

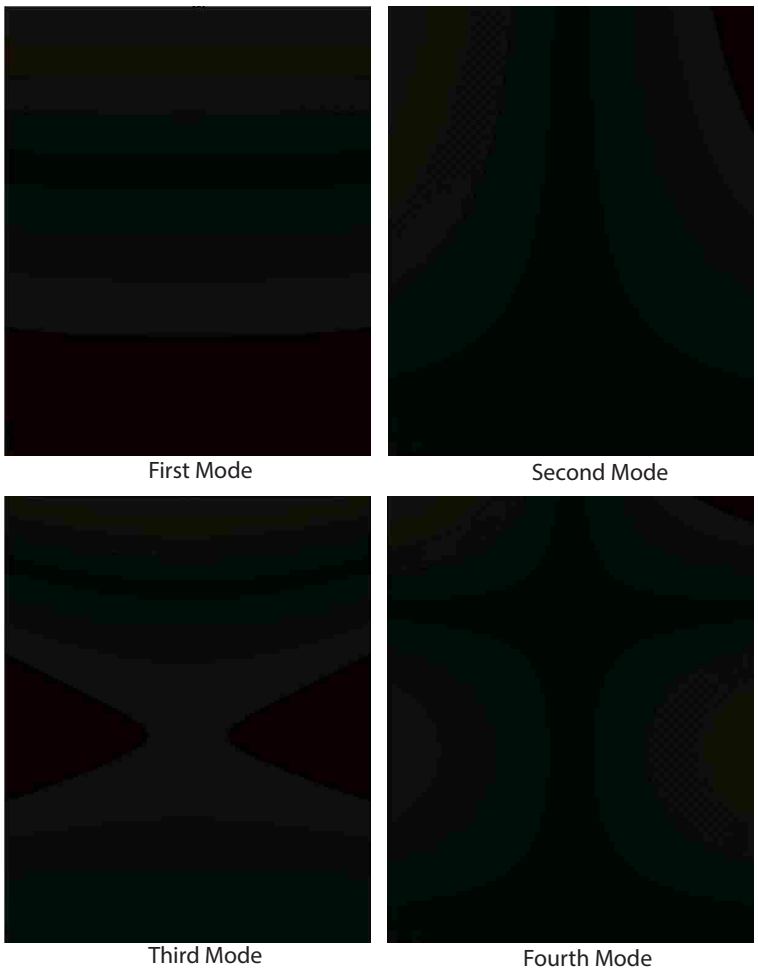

Figure 1.3: Modal Analysis - Modes at Different Natural Frequencies: 2D projections of Figure 1.2. These images represent the contour plot of actual displacement values for the geometry. These images were created by Selin [10].

Modal analysis is often executed inside a design of experiments loop to understand the dynamic responses of new turbomachinery blades as they are designed. Design of experiments (DOE) is a method that allows a designer to iteratively traverse a design space to explore many design and eventually find a satisfactory solution [10, 11]. The design space is a set of parameters and their ranges, that define an object or structure. For example, compressor blades can be defined by their height, chord, thickness, material properties, and other parameters. As these parameters change, so do the dynamic properties including modes shapes and natural frequencies of vibration.

It is important to the designer what modes and frequencies are generated for each blade design. There are certain modes that must be avoided to prevent stresses caused by deformations

that would resonate during the mission of the engine [12]. Tuning is the process of modifying blade geometry to alter the natural frequencies of the blade so that resonance does not occur at the engine's primary operating speeds. By tuning a blade, a designer can minimize the resonance of the blade during operation of the engine, therefore prolonging the lifespan of the airfoil and the entire engine [13]. When hundreds to thousands of designs are computationally generated, each design's modal displacements shapes and natural frequencies must be evaluated. To avoid having a human user from doing simple mode shape classifications repeatedly, computer automated mode shape classification is often implemented [10, 11].

1.2 Thesis Objective

The objective of this research is to develop and evaluate a general method to automatically identify vibrational mode shape images generated from finite element models (FEM) using computer vision and machine learning. Through automating mode shape identification, designers and engineers will receive information about a turbomachinery blade's dynamics behavior more quickly and more accurately.

This work also investigates different algorithms for image warping. Image warping is performed to translate, scale, and rotate an image or parts of an image [14]. When mode shape images are generated they vary in geometry. The warping methods investigated in this work should be able to transform any mode shape image to a standard template to allow for consistent feature extraction, regardless of differences in geometry or mode shape.

In addition, this thesis will report on the speed and classification performance of the mode shape classification method presented. These values will be compared to prior work in automated mode shape identification methods. The main benefit of this research will be to determine if mode shape identification can be performed using vibrational mode shape images from finite element analysis, and if so, can it be done faster and more accurately than other methods. In summary, this research seeks to answer the questions:

- Can modal displacement vectors created within a DOE be identified by their images?
- If so, how will it work?

- How will it compare to prior work in mode shape identification in accuracy and computation performance?
- How does image warping affect the results of mode shape identification? Does it help?

1.3 Problem Constraints

This study investigates a method for combining image processing, pattern recognition, machine vision, and machine learning techniques to automatically identify mode shapes as part of the post-processing of a DOE loop for engineering exploratory design and analysis. By creating this automated process, it removes an engineer from doing manual mode shape identification. In addition, the development of this method will limit mode shape images to four sided blade geometries.

1.4 Thesis Organization

This thesis is organized into six chapters. Chapter 2 is a literature review that introduces modal analysis and prior research in mode shape identification, computer vision, machine learning, and applications of computer vision in engineering and other operations. This chapter concludes with an overview of principal component analysis. Chapter 3 discusses the methods used in this research to perform mode shape identification. First, it describes the steps needed to generate the mode shape images that were used to train and to test the identification algorithm. It also provides some examples of the mode shape images that were used. Second, this chapter describes the methods and algorithms that created the "Eigenblades" classification method. Chapter 4 details the implementation of the methods discussed previously in Chapter 3 and includes listings of Python scripts used to implement the algorithms. Chapter 5 discusses the results of the proposed algorithm at performing automatic mode shape identification, compares it to previous attempts and considers the scope and limitations of the proposed method. Finally, Chapter 6 concludes this work.

In addition to the main chapters of this research, the appendix contains listings of code used to implement the algorithms that are described within this thesis.

CHAPTER 2. BACKGROUND

2.1 Modal Analysis

For mechanical structures, resonant vibration can create highly destructive results as discussed in Chapter 1. Modal analysis has become the standard way of finding the modes' frequencies and their respective vibration shapes. A brief summary of modal analysis will be given here however, more can be found in [9, 15]. A modal analysis solution is found by solving the matrix eigenvalue and eigenvector problem for a mass-springs-damper system,

$$[\mathbf{M}]\ddot{\mathbf{x}} + [\mathbf{C}]\dot{\mathbf{x}}(t) + [\mathbf{K}]\mathbf{x}(t) = \mathbf{f}(t) \tag{2.1}$$

where **[M]**, **[C]** and **[K]** are matrices for mass, damping, and stiffness, respectively. Often, damping is ignored because the characteristic roots and modes between small damping and undamped models are often similar. [16].

Converting to the Laplace domain and with the simplification mentioned above, the equation reduces to,

$$([\mathbf{K}] - \omega^2[\mathbf{M}])\{X\}e^{i\omega t} = 0 \tag{2.2}$$

where ω is the natural frequency and $\{\mathrm{X}\}$ is the modal displacement vector.

Modal analysis is often performed in finite element solvers like ANSYS, where meshed geometric models, given material properties, displacement constraints, and other parameters, are used to calculate the principal modes of vibration.

2.2 Prior work in mode shape identification

One of the first attempts at performing automated mode shape identification used the Modal Assurance Criterion (MAC) to compare two modal vectors. The MAC was developed by Allemang

to provide a measure of equivalence between two modal vectors [15, 17]. Results from the MAC calculation return a value between zero and one, with zero meaning no correlation between the two vectors and one signifying a perfect match. This allowed analysts to compare an analytic model of a mode shape to a set of known experimental mode shape templates and vice versa. One issue with the MAC is that when comparing two modal displacement vectors, both modal vectors must be the same matrix size, $q_1 = q_2$ and $p_1 = p_2$ (where q and p are integers and represent the number of rows and columns in a matrix).

In 2012, Selin [10] developed a method that used the MAC along with nonuniform rational B-splines (NURBS) to classify vibration mode shapes and each mode's corresponding frequency [18]. Selin implemented the MAC on modal displacement matrices that had different mesh sizes by using NURBS to interpolate common points from two different displacement vectors. Selin's proposed method to classify mode shapes resulted in ranges of 78% accuracy for complex, twisted-tapered non-linear plates to 98% for simple rectangular plates (see [10] for full results). Selin noted that because NURBS requires selecting control points from a common grid for interpolation, point selection consistency was often an issue. However, the speed improvement over doing mode identification by hand justified the error.

Porter et al. extended Selin's method in 2016 to use machine learning (ML) to classify mode shapes, rather than using the MAC [11]. Their method also used parametric NURBS geometry to interpolate common points and utilizes ML to classify mode shapes using a large training set. They used the Support Vector Machine (SVM) learning algorithm to train the computer on mode shapes' displacement [11]. This method showed a speed and accuracy improvement when compared to Selin's work. However, there were three issues with this method that resulted in mismatches. First, the lack of training examples for some shapes limited the ability of the system to learn those classes. Porter et al. suggested this could be solved by more training examples per mode shape type and more mode shape sets. Second, there were examples when one mode shape conflated into other mode shapes. They did not give a suggestion of how to resolve this issue. Third, the issue mentioned in Selin's method with NURBS point selection consistency was not resolved.

Similar work by Zang and Liu [19] applied Zernike moments [20] to vibration mode shape identification in aerospace applications. Zernike moments are orthogonal polynomials that can

describe a disk [21]. By using the Zernike moments to perform feature extraction, they showed that finite element modeling and experimental modal analysis data could be classified by their mode shapes. They used the MAC to classify their axisymmetric modal displacement data.

2.3 Machine Learning

Porter et al. recommended utilizing ML algorithms to classify mode shapes. As they state, "Machine learning is broadly defined to include any computer program (or class of algorithms) that improves performance at some task through experience" [11]. This "experience" is a training set: a matrix of numeric values where each row describes an object of similarly related subjects. The three main types of ML algorithms are supervised, unsupervised, and reinforcement learning [22]. Much has been written on ML and its applications; refer to [22] for more information on this topic.

Depending on the type of input, output, speed, and, complexity, there are many algorithms that a designer can choose to implement. Input data is labeled or unlabeled and outputs are continuous or discrete. Three major ML algorithm types are described below.

- Supervised ML algorithms perform classification or regression from labeled training data. Training data is a set of inputs and known outputs. These algorithms use the training set to create a function that maps the input to the output. New data, called test data, is input into the function and an answer mapped by the function is returned. Some supervised ML algorithms are: Support Vector Machine and Bayesian statistics.

- Unsupervised ML also performs classification or regression, but without labeled data. These algorithms find relationships in the data statistically. These algorithms infer the function that maps an input to an output without human interactions. Some unsupervised learning algorithms are: k-Means, nearest neighbor, and clustering.

- Reinforcement ML is a combination of supervised and unsupervised learning. It allows a program to automatically determine the output without supervision. Feedback can be given to a human user about the output to allow the user to adjust the behavior of the learning algorithms in real time [23].

Table 2.1: Examples from the Iris data set. There are four dimensions that numerically describe the Iris type and one dimension that labels the Iris.

X1	X2	X3	X4	Iris type
5.1	3.5	1.4	0.2	I. Setosa
6.0	2.9	4.5	1.5	I. Versicolor
6.0	3.0	4.8	1.8	I. Virginica

Though labeling training examples is an expensive task, it is by far the most accurate type of machine learning because it removes a lot of guess work that the computer would have to do on its own [22]. For the purposes of this work, accuracy is the dominant constraint. Therefore, supervised machine learning was selected.

The data used in machine learning algorithms are often multidimensional. For example, a common data set used to test machine learning algorithms is the Iris flower data set introduced by British statistician Ronald Fisher in 1936 [24]. This dataset describes three types of Iris plants (Iris Setosa, Iris Versicolor, Iris Virginica) using four different values, or more commonly in the machine learning community, features. These four features (sepal length, sepal width, petal length, petal width) can be thought of as occupying a 4-dimensional space with the 5th dimension as the label for a row in the set. Table 2.1 references a few samples from the complete iris data set. Each row represents a particular flower and each column represents the particular feature that describes the Iris flowers.

As data is generated, it can be divided into either a training set or a test set. A human expert can take a small portion of the generated data and manually label them. This labeled data becomes the training set. It is assumed that the human user is perfect at their job and does not mislabel any of the training set data. Using these labeled examples, a machine learning algorithm creates a function that maps the feature vectors of the training set to its label. This function can then identify unknown test data after sufficient training. This allows for future unclassified data sets to be labeled by the function created previously by the machine learning algorithm.

As seen in Table 2.1, there are three possible Iris species. This example data set is a multi-class classification problem, i.e., there is no one line that can separate the three classes. In multi-class classification problems there are two main modes of creating a classifier: one-vs-all or one-vs-one [25]. In a one-vs-all classifier, given n-classes, $n-1$ classifiers are generated and

each classifier is a binary classifier. For example, the Iris data set contains three classes. In a one-vs-all approach, Iris Setosa will be labeled as true, and both Iris Versicolor and Iris Virginica (Iris Setosa vs Versicolor and Virginica) will be labeled as false when training. In this case, a line can be created to divide the two classes. For one-vs-one approaches, there are $n*(n-1)/2$ binary classifiers created. This strategy creates a binary classifier for each possible pair of classes. For the Iris data set, that would create 3 classifiers: Setosa-vs-Versicolor, Setosa-vs-Virginica, and Versicolor-vs-Virginica.

In this research, three supervised machine learning algorithms were evaluated on how well they would classify mode shape images. The three algorithms, neural networks, *k*-nearest neighbors, and support vector machines are briefly presented here along with their advantages and disadvantages. Though there are other types of classical machine learning algorithms, these three gave better results than the rest for the classification problem of identifying modes shapes (other machine learning algorithms that were tested include: *k*-means, decision trees, and Gaussian methods).

2.3.1 Neural Networks

Neural networks began their development as an attempt to copy the human nervous system [26]. In the human brain there are billions of neurons that are interconnected through synapses [22]. Their general operation use transmitter chemicals within the brain that raise or lower their electrical potential on thought, stimulus, or other actions internal or external to the body. A neuron fires (is activated) when its electrical potential reaches some threshold. The activation of interconnected neurons is how the brain transfers information, makes calculations and generally creates the sentient, learning being. Though the understanding of the neuron is limited, the research for neural networks as artificial learning algorithms quickly evolved to allow multidimensional data to be used for linear regression and classification.

One of the first neural networks developed that mimics biology was the perceptron neural network learning algorithm [27]. This algorithm is graphically shown in Figure 2.1. Given some inputs (x), weights (w), learning rate (η), target value (t) and the threshold value for multiple neurons, this neural network can be trained to solve linear problems [22]. The goal of neural networks is to change the weights of each neuron until the target value (t) is the same as the

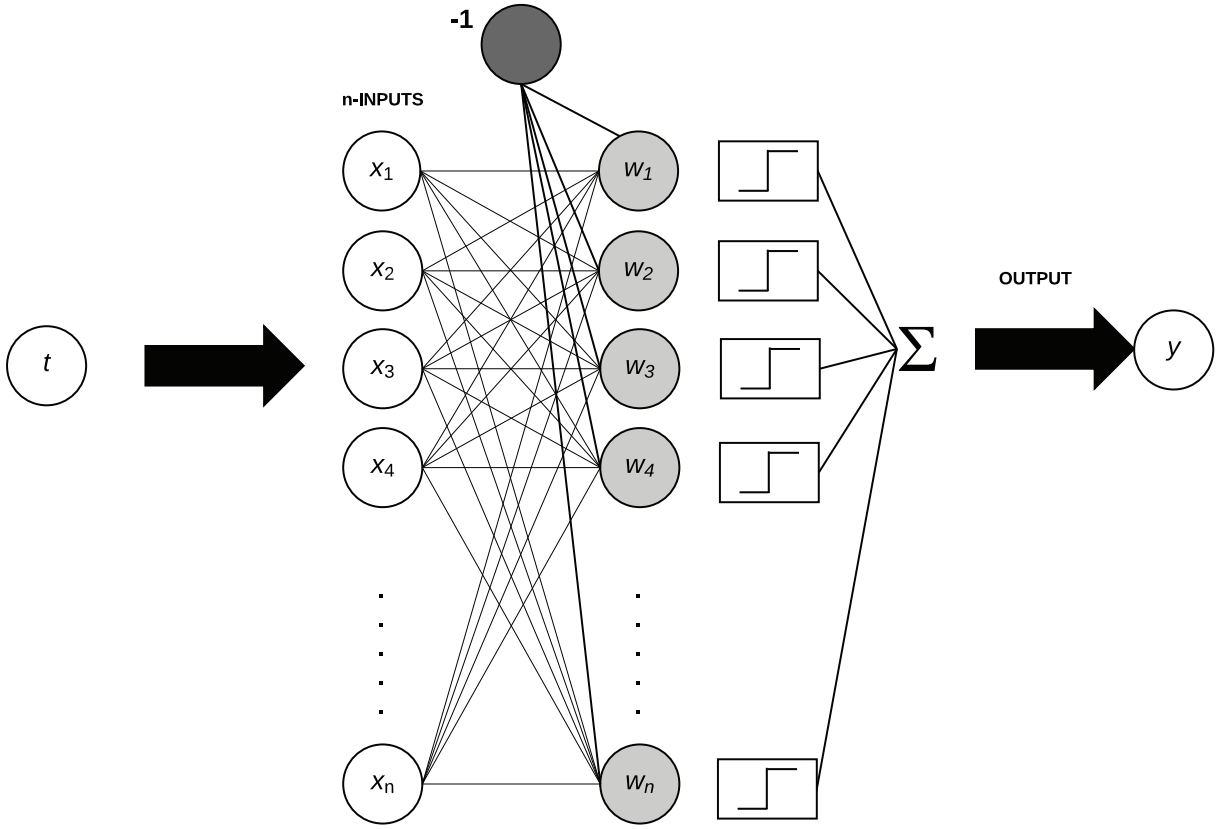

Figure 2.1: Perceptron neural network learning algorithm. Elements of the perceptron include: t - the target value of the example, x - the input values of the example, w - the weights of the neurons, the bias weight (-1), the threshold mechanism for every neuron, the summation, and the output value [22].

output value (y) for every example within the training set or until a set number of weight change iterations have occurred. More layers of weights could be added to gain the ability for more complex learning [28].

The value of -1 is called the bias value. It is an input value that is automatically taken into account for each calculation because it allows for a change within the weights even when the inputs (x) are 0. This allows the network to learn even in the null case.

The governing equation to change the weights is

$$\Delta w_{ij} = -\eta * (y_i - t_j) * x_i \tag{2.3}$$

The value η is the learning rate. This value decides how rapidly the network learns. A large value of η allows the network to change weights quickly, thereby allowing it to learn quickly. However, that could create an unstable network because it never allows the weights to reach a steady state. A small value of η would allow the learning to be precise, however it will take too long. Typically a value of $0.1 < \eta < 0.4$ balances the two competing objectives.

The perceptron algorithm is as follows and is taken from [22]:

- **Initialization** set all of the weights w_{ij} to small random numbers
- **Training**
  - for T iterations or until all of the outputs are correct
    - for each input vector **x**
      - compute the activation of each neuron j using the activation function g

$$y_j = g(\sum_{i=0}^{m} w_{ij} x_i) \tag{2.4}$$

        where $y_i = 1$ if $g > 0$ and $y_i = 0$ if $g \leq 0$

      - update each of the weights individually using equation 2.3 where it is repeated in equation 2.5 for convenience.

$$\Delta w_{ij} = -\eta * (y_i - t_j) * x_i \tag{2.5}$$

- **Testing** - finding the answer with unknown input data
  - compute the activation of each neuron j using

$$y_j = g(\sum_{i=0}^{m} w_{ij} x_i) \tag{2.6}$$

    where $y_i = 1$ if $w_{ij} x_i > 0$ and $y_i = 0$ if $w_{ij} x_i \leq 0$

This algorithm uses a training set's labeled data to iteratively change the weights for T iterations or until all of the target values (t) match with the output value (y) for its inputs (x). The

weights that are generated are saved and can be used to test/classify unknown data. This algorithm only provides two output values, 0 and 1. In this algorithm, the threshold value was 0. This means a value less than or equal to 0 does not activate the neuron. Using equation 2.3 for each feature vector, the weights change with each training example that does not output the target value.

In the case where there are more than two classes, a one-vs-one strategy needs to be employed [29] as discussed previously. This strategy creates a new perceptron classifier for each combination of two different classes. Each of these perceptron classifiers are then trained using their corresponding training examples. Given three classifiers, when an test example is presented for classification, each classifier runs the input data and the Iris type with the most votes wins, and the input data is then labeled.

The perceptron was one of the first machine learning algorithms to be created and used for classification and regression. It was used in an application of atmospheric science where it predicted the air quality of industrialized urban areas, sulfur dioxide concentrations, and even predicting weather [30]. However, this algorithm is limited in that a singled layered perceptron only can create linear breaks between classes [22]. The multi-layered perceptron can create a sigmoid function [30], which would allow for more complex representations of class division lines. A limitation with perceptron learning algorithm is that though it calculates a line that separates the classes, it is often not the best line that maximizes generality. In most cases, the algorithm runs until all of the training instances targets match their output, and there is no guarantee the line created is the best solution.

2.3.2 k-Nearest Neighbor

One of the simplest classical supervised machine learning algorithms used today is k-nearest neighbor [31]. This algorithm is visualized with Figure 2.2. The classifier is trained by using every training example and saving them into a database. To classify an unknown test example, the new point is input into the classifier and then compared to every point within the training set. The k closest neighbor's labels are then used to classify the unknown example by a simple majority rules. In the example shown, there are a total of 25 training examples: 9 red triangles, 9 green squares, and 9 yellow circles. The unknown shape is labeled as a star. Choosing k to be 5, one can see that the majority of the shapes are red triangles because of the 5 nearest neighbors, and

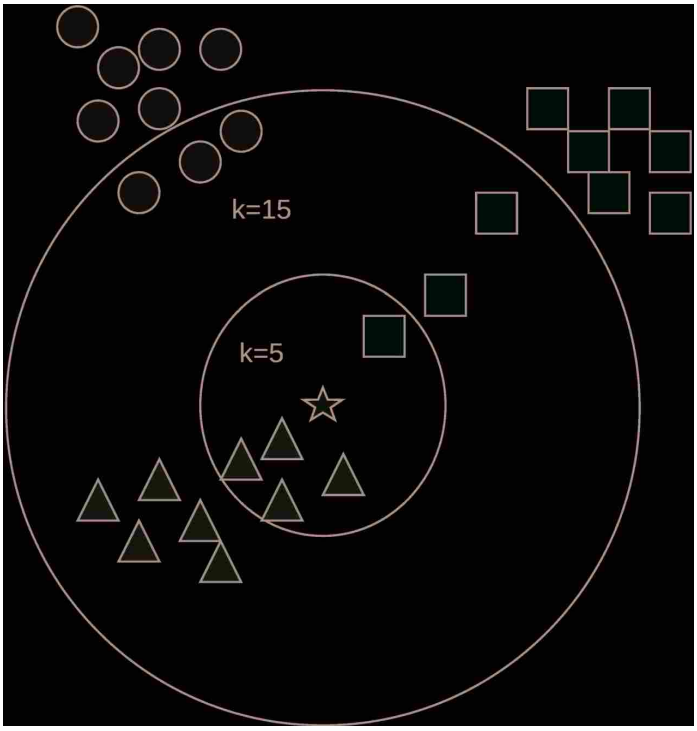

Figure 2.2: k-nearest neighbors learning algorithm. The unknown example is the gray star. Given k to be 5, there are 4 red triangles and 1 green square that are the nearest neighbors. By a simple majority vote, the unknown shape is classified as a triangle. Also, when k is chosen to be 15, again the red triangles win the majority and the unknown gray star is labeled as a red triangle.

the triangles win by a majority. The same conclusions can be reached when k is 15. An odd value for k is often used for tie-breaking.

Though the algorithm is easy to comprehend and implement, there are some limitations. First, the entire training set must be used and kept in memory. Given an unknown test example, this point must be compared to all of the surrounding points that were given in the training set. This comparison could be sped up with the use of good coding practices and binary trees, but a large amount of comparisons must still be made [32, 33]. In addition, the value of k must be chosen by a human user. For 2D or 3D data that could be visualized, this would be a simple practice. However, with more than 3-dimensions, it would be difficult to choose k without experimentation. In Figure 2.2, it can be seen that the value 5 for k will be a good value to choose. Given k to be 11, though the outcome is the same, more time would be needed to perform the computation. This would scale with the number of features/dimensions. With large data sets or data with high dimensionality, the distance computations and comparisons are very computationally expensive.

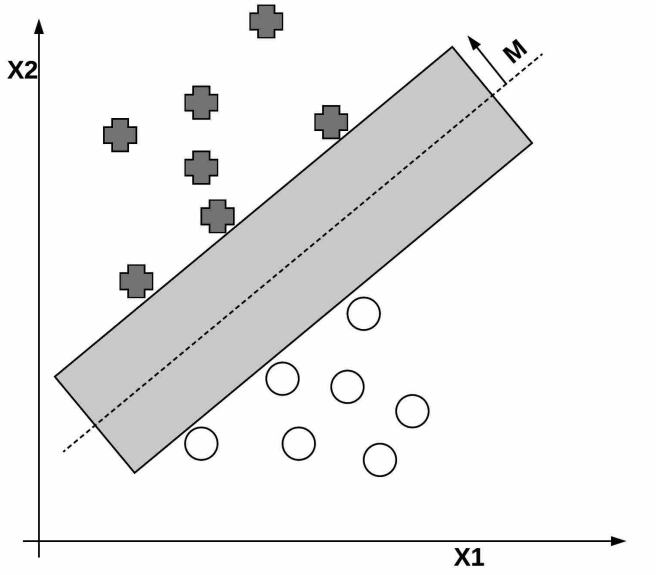

Figure 2.3: Graphical motivation for Support Vector Machines. Any separating line between the two classes works well to create a classification boundary, as seen by the gray rectangle. However, to get the best generalization, a line that maximizes the margin, M, is calculated in SVM. In this case, the best line is the dotted line exactly between the two classes. The +'s and circles touching the gray rectangle are the "supports" of the classifier.

2.3.3 Support Vector Machines

Support vector machine (SVM) is a type of supervised learning algorithm which seeks to maximize the margin, M, as shown in Figure 2.3. In this example, a class is described by two features, $X1$ and $X2$. These two values describe the class and are labeled as either X's or O's. More than two dimensions in real data sets are extremely common. Notice that any line within the gray area can separate the classes correctly. However, the line in the center (the dashed line) of the box should be used to achieve the best generalization of the problem. This gray box can be visualized as a cylinder in 3D and a hyper-cylinder in higher dimensions. The margin, M, is the radius of the cylinder. The purpose of SVM is to find a line in a multidimensional problem that maximizes the margin between two classes. A brief summary will be given here. Refer to [22] for detailed information.

The data points that lie closest to the classification line are call the support vectors. These points define the size of the margin. A convenient feature of SVMs is that once training is complete,

only the support vectors are needed to define the separating planes. The rest of the data can be thrown out to save memory.

Given that a classifying line can be defined with the equation $\mathbf{w}^T\mathbf{x}+b$ and the margin M, one can see that any input point $\mathbf{x}$ where $\mathbf{w}^T\mathbf{x}+b \geq M$ is a positive example, as seen in Figure 2.3. Similarly, a point where $\mathbf{w}^T\mathbf{x}+b \leq -M$ is classified as a circle. Given the classifier line $\mathbf{w}^T\mathbf{x}+b$, the next step is to find the largest possible value for the margin M. This is the same as making $\mathbf{w}^T\mathbf{w}$ as small as possible. Mathematically, this is equivalent to minimizing $\frac{1}{2}\mathbf{w}^T\mathbf{w}$.

This then turns into a quadratic optimization problem with linear constraints [22]. This type of optimization problem must be solved using the Karush-Kuhn-Tucker (KKT) conditions:

$$\lambda_i^*(1-t_i(\mathbf{w}^{*T}x_i+b^*))=0, \tag{2.7}$$

$$1-t_i(\mathbf{w}^{*T}x_i+b^*)\leq 0, \tag{2.8}$$

$$\lambda_i^* \geq 0, \tag{2.9}$$

where $\mathbf{w}$ is the weight vector, similar to the slope of a linear equation that the algorithm is trying to learn, b^* is the bias value similar to the x-intercept of a liner equation, x_i is the input feature vector of what the algorithm is trying to classify, and t_i is the target for the current feature vector.

The variables λ_i are positive values known as Lagrange multipliers. A non-zero value means that they are active constraints. Given these constraints, the final minimization problem can be defined in Lagrangian form as:

minimize:

$$Ł(w^*,b^*,\lambda)=\sum_{i=1}^{n}\lambda_i-\sum_{i=1}^{n}\lambda_i t_i-\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\lambda_i\lambda_j t_i t_j x_i^T x_j \tag{2.10}$$

In addition to linear problems, non-linearly separable problems can also be solved using slack variables and different kernels. More information on the implementation is given in Marshland's book and resources found elsewhere [22].

2.4 Computer Vision

Often computer vision (CV) works with ML to recognize or classify patterns within images. Images are 2-dimensional representations of the 3-dimensional world. Examples of CV applications include: optical character recognition, machine inspection, and facial recognition. Since computers do calculations in discrete values, arbitrary qualities of images like color, form, and shape are difficult to quantify. Researchers in CV seek to create algorithms that quantify images as discrete values for algorithmic operations like comparison, matching, and classification. These discrete values are often n-dimensional arrays called feature vectors. This field of research is call shape classification. A survey of shape descriptor algorithms can be found in [34] and [35]. Examples of shape descriptors are: area, eccentricity, moments, and center of mass [35]. In this research, computer vision will be used to extract the feature that describe a mode shape image for futher recognition.

2.5 Applications of computer vision and machine learning in identifying objects

The work described so far focused specifically on how to identify mode shapes from displacement data from either models or experimentation. Wang et al. introduced the idea of using images, pattern recognition, and image processing to classify mode shapes for engineering applications. Their work in [36–39] used mathematical transformations of images and experimental data to perform mode shape identification. These techniques decomposed mode shapes, whether from a model or from an experiment, into a feature vector. In other work, they implemented feature extraction through Zernike Moments [20], Fourier Descriptors [40], and Wavelet Descriptors [41], studying the effects of using these methods on both circular and rectangular plates. Classification is then done using different approaches such as nearest-neighbor or k nearest-neighbor. Their work resulted in high accuracy with both analytic and experimental models.

Much work has been done in pattern recognition within the computer/machine vision community. A landmark algorithm in facial recognition is Turk and Pentland's "Eigenfaces" approach [42], which demonstrated how to decompose facial images into their principal components to perform classification and image reconstruction using principal component analysis (PCA). It should be noted here that often in PCA, the principal components are called the principal modes of

variations, often shortened to the "modes" of a data set. Throughout this paper, the term "mode" or "modes" will refer to the principal vibrational displacements on turbomachinery blades images. The principal components will be stated directly. Belhumeur et al. [43] later extended the work of Turk and Pentland by again learning statistical variation in facial appearance in their "Fisherfaces" approach, which uses Fisher's linear discriminant to extract features for nearest-neighbor classification.

Modern pattern recognition and computer vision applications have used convolution neural-networks to perform classification, which has grown popular because of its results and deep learning abilities [44–46]. However, there is current research in applied recognition which continues to use the classical approach of feature extraction as well as other forms of machine learning to improve classification accuracy. Experiments in other applications such as the work done by Mozina et al. [47] demonstrated how automated visual inspection can be done on pharmaceutical tablets. Mozina et al. showed how to check the three-digit imprint on a tablet with good results compared to the manual inspection and prior work [47]. Larese and Granitto [48] used scale-invariant feature transformation (SIFT) and SVM to classify different legume species. In their work, they locate unique identifiers on legume vein pattern structures for classification. Their work produced excellent results which outperformed previous literature and manual expert classification. Other work done by Biagio et al. [49] applies image processing and machine vision to inspecting aerospace systems. In their work, they set up an experimental camera and lighting rig with template matching software. This setup was used to demonstrate the effectiveness of their model checking and part visual inspection algorithms. In their work, model checking was used to inspect CAD models to automatically identify if they meet predefined specifications, and part visual inspection was used to check the surface finish of physical parts. Using linear support vector machines as their classifier, they found on average an accuracy of 95.22% for part identification and 67% for part visual inspection.

In addition, other work in pattern analysis, image processing, and computer vision has shown improvements in identification accuracy by preprocesing images prior to feature extraction. Work by Usman and Rajpoot [50] proposed a brain tumor segmentation and classification method for multi-modal MRI scans. In their work, they used an expert to skull-strip the image to isolate

the brain, then used a mask and crop technique to extract a bounding box of a tumor from the rest of the image.

Work done by Giraldo-Zuluaga et al. [51] automatically identified a species of micro-algae from microscopic images. These images were often blurred, non-homogeneous, or had salt and pepper noise that affected the performance of their classification algorithm. In their work, they proposed using a segmentation algorithm which included Otsu's binarization [52], contour finding, Sobel filters [53], and Fourier transforms [54] to filter, orient, and subsequently extract the micro-algae profiles from the main image.

Khan et al. [55] presented an algorithm to increase sensitivity of segmenting retinal vessels in human eyes. Linear morphological operators like top-hat transformations [56] were used to segment the image. In addition to using homomorphic filtering with high pass filters [57], they used other techniques like calculating the second order Gaussian derivative and oriented diffusion filtering [55] to get a high quality segmentation of the retinal vessels.

2.6 Principal Component Analysis

Machine learning algorithms take training example data sets and create a function that maps the input feature vectors to the output label, which can then classify unknown test data. For the Iris data set mentioned previously, there were only five dimensions that describe an example, four that described the measurements of the sepal and petal, and one that labeled the type of Iris plant. However, in large n-dimensional data sets like an image, this could be a problem. For example, given an image of a cat that is 5360×3560 pixels, this translates to a $19,081,601$ dimensional space, where each pixel location is a value that describes the color of the pixel. In this case, one dimension is the label (cat), and the rest define the pixels. Given five images (3 cats and 2 dogs) of the same pixel size, this creates 5 points in a $19,081,601$ dimensional space. One can see this is an expensive task when calculating the maximum margin classification line for these images when using SVM. The task then is to reduce the number of dimensions while generating a feature vector of the images that keeps distinctive information in each image.

This is where dimensionality reduction algorithms like principal component analysis (PCA) work well. This research leveraged the work done by Turk and Pentland in their "Eigenfaces" paper [42] to recognize and classify human faces. In Turk and Pentland's paper, principal component

analysis was used to extract the eigenvalues and eigenvectors of a covariance matrix of a set of images [42]. Using the "Eigenfaces" paper as inspiration, in this research the eigenvectors are referred to as the 'blade-space'. A brief introduction of PCA will be given here. More information can be found in [58]. After performing PCA on the training images, the original training images are projected back onto the 'blade-space' by an inner product. The number of principal components, P, is chosen to create a representative subspace created from the training set. When a projection is performed, an array of length P is returned, which is called the PC-score for an image. This array can be used as a descriptor for the image.

More specifically, let $\mathbf{I}(x,y)$ be a two-dimensional $N \times M$ array of 8-bit (gray-scale) intensity values. Supposed each image is a $N = 100$ by $M = 100$ pixel images. This then means each image is represented by a point in a 10,000-dimensional space. PCA is used to reduce the number of dimensions needed while keeping the descriptive information of each image.

Let there be a set of n, 2-D training images, $\mathbf{I}(x,y)$. Each image is flattened to a column vector so it is contained in the space of dimensionality NM. The mean blade image is then defined in the traditional manner,

$$\mathbf{m} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{I}_i \tag{2.11}$$

Each blade image is mean adjusted so that $\mathbf{x}_i = \mathbf{I}_i - \mathbf{m}$. A data matrix is created where $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots \mathbf{x}_n]$ of dimension $NM \times n$. The covariance matrix, $\mathbf{C}_x$ of the training image set is:

$$\mathbf{C}_x = \mathbf{X}\mathbf{X}^T \tag{2.12}$$

Given the covariance matrix, this then becomes an eigenvalue problem,

$$\mathbf{C}_x\mathbf{U} = \mathbf{U}\mathbf{L} \tag{2.13}$$

where $\mathbf{L}$ is a diagonal matrix of the eigenvalues, λ_i, of the covariance matrix, and $\mathbf{U}$ is the set of associated eigenvectors for the eigenvalues. Each eigenvector $\mathbf{u}_i$ is a row of $\mathbf{U}$, which then represents a principal axis where the data can lie. While solving the eigenvalue problem, the eigenvalues are often sorted by their magnitudes along with their associated eigenvector. The dimensionality reduction then comes from choosing a value $P < N \times M$, and taking the P most

significant values of $\mathbf{L}$ and the corresponding eigenvectors from $\mathbf{U}$. This subset space is called the PC-space and is represented by $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$. The subset $\hat{\mathbf{U}}$ (of size $P \times NM$) is then used to reduce each image.

Given an image, $\mathbf{I}$, which is first mean adjusted to become $\mathbf{x}$, it is then projected onto the PC-space formed by the training images' by the inner product equation,

$$\mathbf{y} = \hat{\mathbf{U}}^T \mathbf{x} \tag{2.14}$$

where $\mathbf{y}$ is a vector of length P, which represent the principal component (PC) scores for the image $\mathbf{I}$. These PC-scores represent the feature vector that describes the image in the PC-space.

Equation 2.14 then is used to extract the feature vector of all the training images. This is done by projecting each training image onto the eigenspace and then extracting the PC-scores for each image. Given that there are n images, the training data matrix becomes, $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_{n-1}, \mathbf{y}_n]$.

In addition to feature extraction, principal component reduction techniques also return the eigenvectors for each image. Each of these eigenvectors represent a portion of a linear combination of all the eigenvalues that represent the original sample. Therefore, adding all the principal components eigenvectors of an image together will reconstruct the image. This work used the Python library `sklearn` implementation of PCA and the components discussed [59].

CHAPTER 3. METHOD: THE EIGENBLADES CLASSIFICATION METHOD

3.1 Generating mode shape images

The first step in the Eigenblades method is to gather the images, as shown in Figure 3.1. Prior to this research, there was no large repository of mode shapes images that could be used as training or test sets. Therefore, generating mode shapes images is the first task. Mode shape images were generated using a design of experiments (DOE) work-flow as shown in Figure 3.2.

The high-level method begins with describing the input files used for defining blade geometry used in the DOE. The parameters that define geometry of the turbomachinery blades are defined in this file. The theory of Latin-hypercube sampling is then presented. After that, a brief summary will be given of using NX and the NX OpenAPI to create solid geometry. A discussion of using Mechanical ANSYS to perform modal analysis on blades and then extracting mode images is presented next. Finally, some post-processing matters are brought to the readers attention for consolidating data and presenting it in a usable way for the next steps in the Eigenblades classification method.

3.1.1 Design of Experiments Setup

A design of experiments (DOE) refers to the process of planning, designing, and analyzing an experiment so that valid and objective conclusions can be drawn efficiently and effectively [60]. For turbomachinery blades, using a DOE enables more efficient design exploration. Prior work by Selin and Porter et al. [10, 11] used a DOE to create different blades with varying geometry to test their mode identification methods. In this work, the DOE will be used in a similar fashion to generate mode shape images. Figure 3.2 shows a DOE with necessary steps of exploring the design space, changing the parameters, and eveutall generating mode shape images.

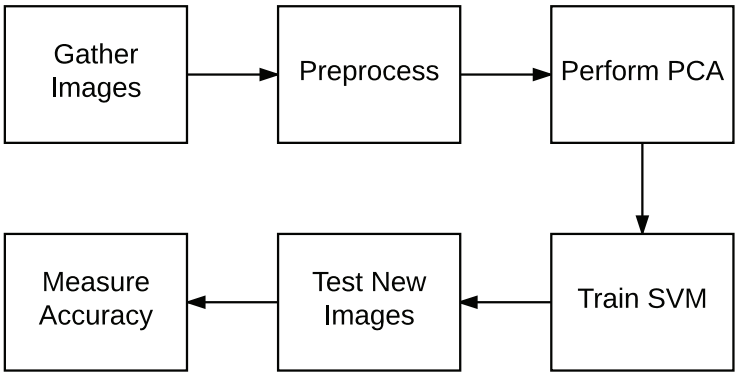

Figure 3.1: High-level view of the "Eigenblades" classification method. The process includes gathering the training and testing images, preprocessing the images, feature extraction with PCA, training a classifier, and then running the classifier on the test images.

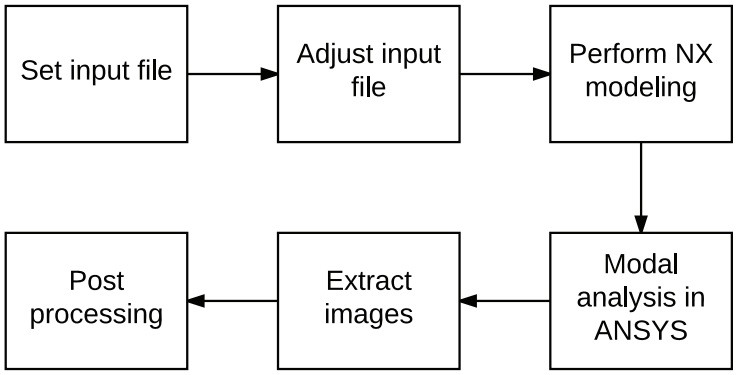

Figure 3.2: High-level view of how to generate mode shape images, which is the first step from Figure 3.1. This begins with choosing the airfoil input file. The file includes the point geometry and the spatial parameters of a selected airfoil. Then the design of experiment (DOE) changes the parameters defined previously by the input file. This input file in used by the NX program to generate solid geometry. Modal analysis is then performed in Mechanical ANSYS and the images are extracted. Finally, some post processing is done on each experiment to consolidate the generated files.

The external file used in this DOE defines the governing parameters, is called the input file. This input file describes the number of sections within the blade, the parameters used, the airfoil file that describes each section's airfoil shape, and the changes in each section's airfoil. All of these parameters govern the geometry of the entire blade. During the design of experiments, the parameters that govern the blade geometry are changed to allow for a variety of designs to be explored.

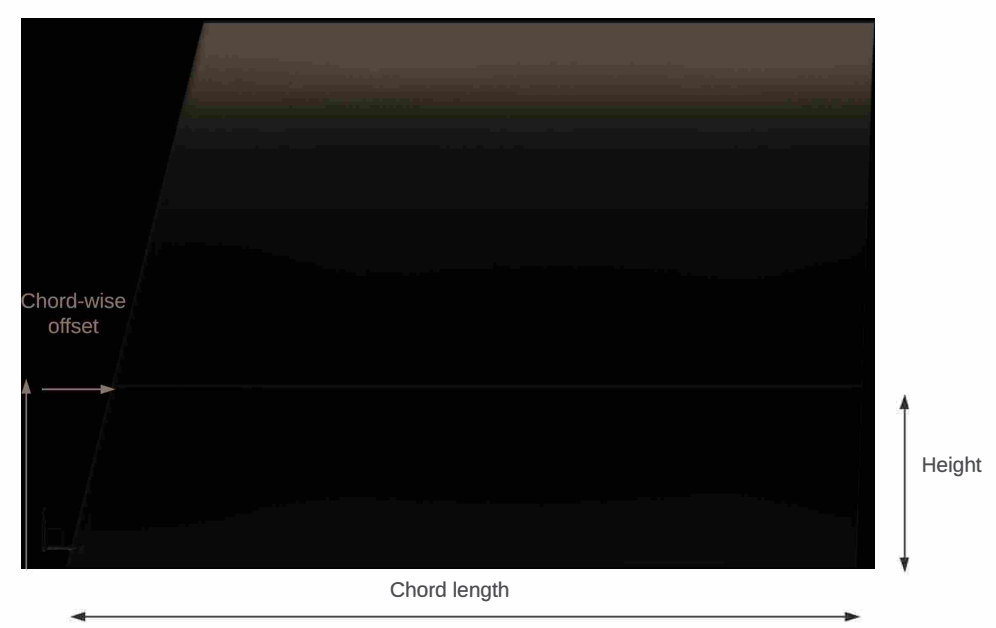

Figure 3.3: Changes of geometry for an example blade's chord, height, and chord-wise offset. In this example, all four sections have a different chord length and chord-wise offset.

3.1.2 Blade Geometry

The structural models used for this work were publicly available aerospace application blade shapes selected from the National Advisory Committee for Aeronautics (NACA) airfoil database. The parameters that govern the turbomachinery blade's geometry were: Chord Length (measured in unit length), Twist (measured in radians, the axial rotation from base to the tip) Chord-wise offset (measured in unit length), Fore-Aft Offset (measured in unit length), and Height (the length from base to tip of a blade, measured in unit length). These values are allowed to change for each section within the blade and will affect the 3D geometry of the blade. The following figures depict how the parameters change the geometry of a blade. Figure 3.3 shows the chord-length, height, and chord-wise offset parameters. Figure 3.4 depicts the four sections of the example blade and the fore-aft offset for the sections. Finally, Figure 3.5 is an extreme twist value on section four (tip of the blade) compared to section one (root of the blade).

Figure 3.4: Blade geometry example shown in sections and fore-aft offset. In each section, the airfoil geometry have been offset in the fore-aft direction compared to the base.

3.1.3 Latin-Hypercube Sampling

Latin hypercube sampling [61] is often used within a DOE to create near random, statistically diverse data points for n-dimensions. It is explained graphically in Figure 3.6.

This example has two dimensions, $x1$ and $x2$, that describe the design space. The goal of Latin-hypercube sampling is to choose values in the ranges of each dimension in a way that is space filling. This allows for a more complete understanding of a design space, while minimizing the number of points that need to be sampled. It can be seen here that each row and column is uniformly explored, even though the entire space has not been sampled. This allows for statistical diversity while minimizing the computing time needed to evaluate all possible points in the design space. Latin-hypercube sampling sets are pre-built for a particular number of dimensions and dimension ranges and are often implemented in pre-built classes for many programming languages, thus allowing for practical automation.

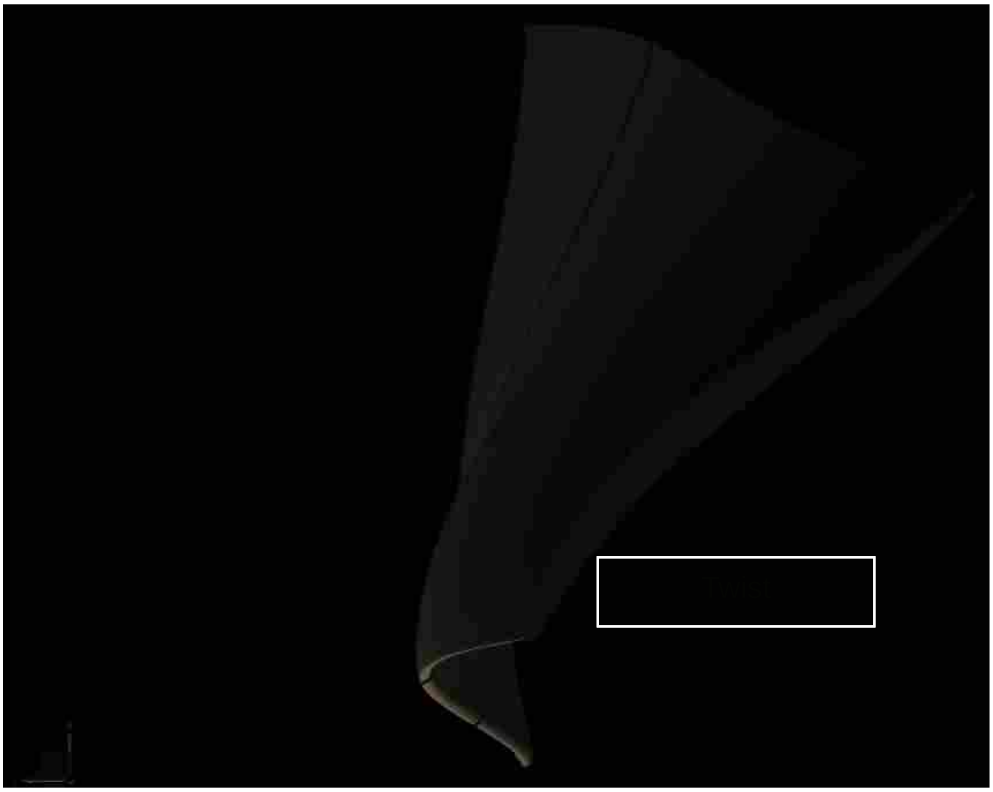

Figure 3.5: Top-down view of the example blade geometry. This blade demonstrates the extreme twists possible for each section of a blade. In this case, section four is rotated 45 degrees clockwise from the base.

3.1.4 NX Geometry

As random samples are created using the Latin-hypercube, each sample defines the parameters that needs to be created in solid geometry. 3D geometry is created within Siemens' NX, a 3D computer aided design (CAD) modeling software. A feature of NX is it application programming interface (API), NX OpenAPI, which allows programmatic interaction between the NX modeling applications and external programming languages. As the geometry is changed during the DOE, a program was built using the NX OpenAPI to create solid geometry for each designed experiment sample without the need of a human user inputting manual operations. Using the NX OpenAPI then allows for an automated system to created 3D blade models.

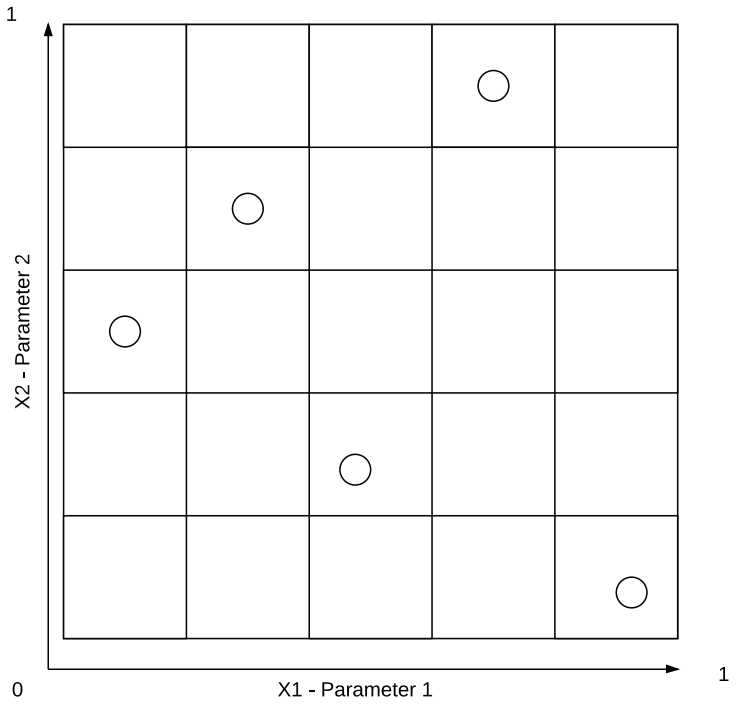

Figure 3.6: Latin hypercube sampling example. Notice that the sample points are spread so that each column and row only has a single sampled point.

3.1.5 Modal Analysis in Mechanical ANSYS

Modal analysis was described earlier as a eigenvalue/eigenvector problem that was derived from a multi-degree of freedom mass-spring-damper system. Finite element packages like Mechanical ANSYS (which was used in this work) provide a finite element model (FEM) approach to performing modal analysis and calculating the principal modes of vibration and their natural frequencies. In these models, solid geometry is tessellated, given material properties, and loading conditions at points and boundaries of the solid model. Then, using the finite elements analysis (FEA) solvers built into Mechanical ANSYS, the mode shapes and their frequencies can be calculated. Like Siemens NX, Mechanical ANSYS also provides an API called ANSYS Parametric Design Language (APDL). This API allows a FEA to be called from external code and does not need a user to manually create the analysis for each different blade geometry that was generated during a DOE, thus, allowing for automated modal analysis to be performed. In addition to performing modal analysis, ANSYS also can capture images of the calculated mode shapes in gray-scale. Modal analysis was performed using the Mechanical ANSYS 17.0 software package.

The models in ANSYS were built using ANSYS's Structural Mass, Solid Tet 4, 285 node elements, which are three dimensional meshes. Modal displacement solutions were calculated using the Block Lanczos [62] algorithm as implemented in ANSYS.

3.2 Preprocessing

After generating the training and test mode shape images, they are preprocessed prior to feature extraction. Images are preprocessed to extract the mode shape from the overall image by removing the white space, and converting the relevant blade geometry to a standard shape. Removing white space and image processing methods are often used to cleanup, or remove noise from the main focus of an image [14]. Since there is no way to account for possibly infinite variations that could occur with a blade's parameters during separate DOEs, every mode shape image is transformed to a 100×100 pixel template square. Every training and test image is mapped to this template square prior to feature extraction by PCA. Two methods for preprocessing were explored in this research, bilinear warping and triangle interpolate warping.

3.2.1 Bilinear Warping

The first preprocessing algorithm investigated was the bilinear warping algorithm. Figure 3.7 shows the steps for this method. First, the corners of the mode shape are found using the Shi-Tomasi corner detector [63]. Given a maximum of four corners per image, the next step is to label the corners as top-left, top-right, bottom-left, and bottom-right. This facilitates a projective transformation from a 4-cornered quadrilateral to a 100×100 pixel size square. This transformation is based on the bilinear warp algorithm [64], which calculates the coefficients needed to map one quadrilateral to another. These mapping coefficients are found using:

$$\begin{aligned} x &= a_0 + a_1 u + a_2 v + a_3 uv \\ y &= b_0 + b_1 u + b_2 v + b_3 uv \end{aligned} \tag{3.1}$$

where (x, y) represents a corner point on a square and (u, v) represent a corner on a quadrilateral. Given four points on a square and its corresponding corners on a quadrilateral, the bilinear equation is used four times to set up a system of equations to solve for the eight transformation coefficients:

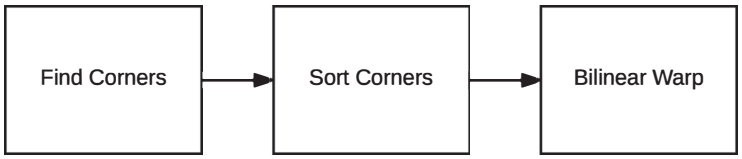

Figure 3.7: High-level preprocessing algorithm for bilinear image warping. The purpose of preprocessing each image is to scale each image into a standard shape and to remove as much white space as possible so only the blade is used for feature extraction.

a_0, a_1, a_2, a_3, b_0, b_1, b_2, and b_3. After the coefficients are found, the warp is performed on each pixel using Equation 3.1.

In addition to scaling and removing white space, through resizing the images into 100×100 pixel images, it reduces the processing time during feature extraction and classification. After warping, any remaining white pixels, or pixels with a value greater than 253, are replaced with a neutral-gray value of 127. This is done to make sure any white pixels remaining from the preprocessing is replaced with the median value of the whole image.

After the image has been transformed to a square, the matrix is flattened into a vector. Thus, each mode shape image becomes a vector of length 10,000, with each entry that has a value between 0-252. The image processing algorithm used were implemented in Python using the OpenCV library [65].

3.2.2 Triangle Interpolate Warping

The triangle interpolate warping method was developed in this research as another method to preprocess the mode shape images prior to feature extraction. Figure 3.8 outlines the preprocessing steps used. Images are preprocessed to extract the mode shape from the overall image by removing the white space, and converting the relevant blade geometry to a standard shape. Again, it is desired that each image is transformed into a 100×100 pixel image to account for possibly infinite variations of blade geometry. Every training and test image is mapped to this template square prior to feature extraction by PCA. The method is best described while using images. The overall method is shown in action in Figure 3.9 and is implemented as follows.

Processing and edge finding: This process begins by reading a mode shape image as a gray-scale image. Afterwards, a Gaussian blur [66] and then a morphological open operator [67]

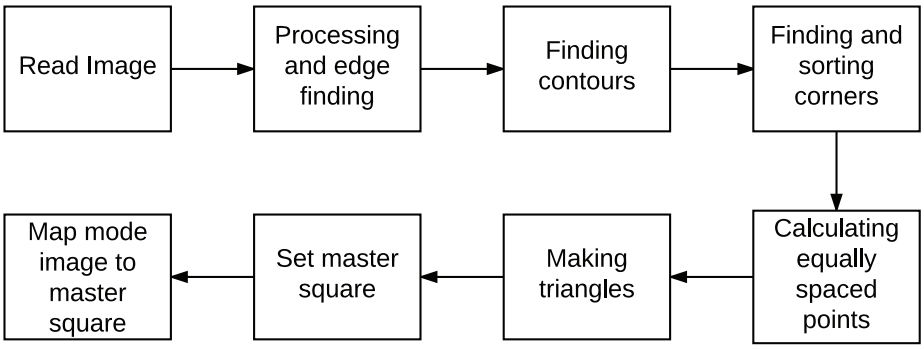

Figure 3.8: High-level implementation of the triangle warping preprocessing method. This method maps triangles from the original mode shape image onto a template square.

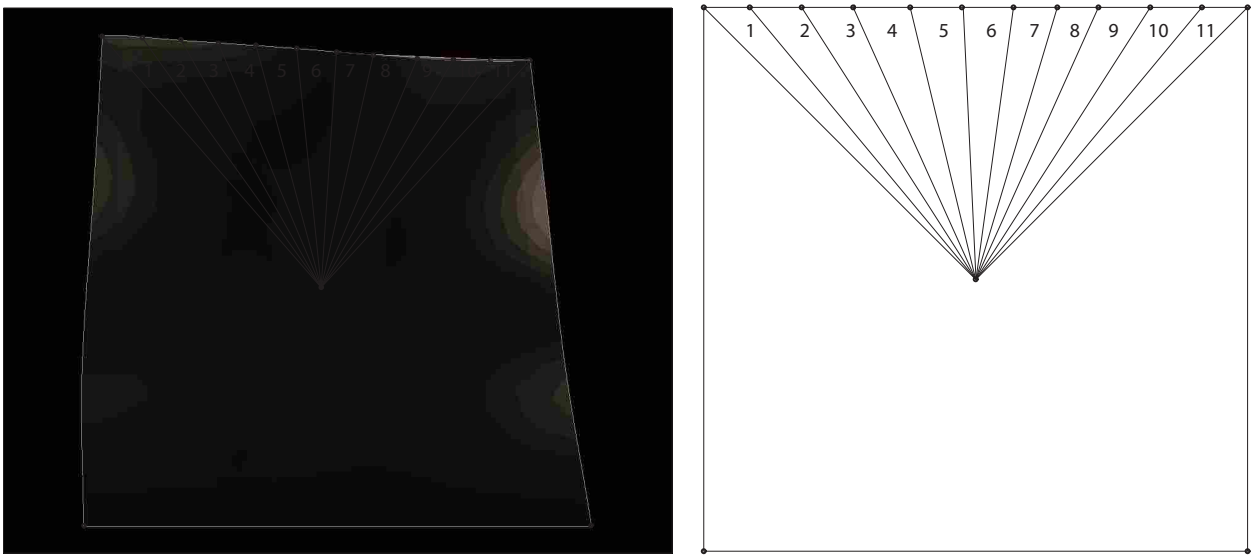

Figure 3.9: Visual explanation of the triangle warping method. First, the edges are calculated and then used to find the outer contours of the mode shape. These contours are then used to extract the corners of the shape. The contour data is then used to create equally spaced points around the mode shape. The center point is also found from the contour data by the centroid of moments. Triangles are then created from the points and then mapped to a template square using an affine transformation for triangles.

are applied to remove any artifacts that are left over from image generation. In addition, these operations smooth the image by removing sharp edges. An example of the blurred gray-scale image can be seen in Figure 3.10. These artifacts tend to becomes false edges or corners. When new DOEs are performed, often the grayscale representation of the modal displacements differ from one experiment to the next. To make sure edges can be found, each new DOE's mode images

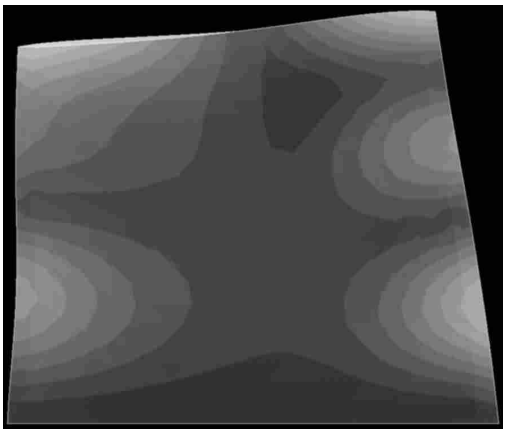

Figure 3.10: Mode shape image as a gray-scale image and applying the Gaussian blur and morphological open operation.

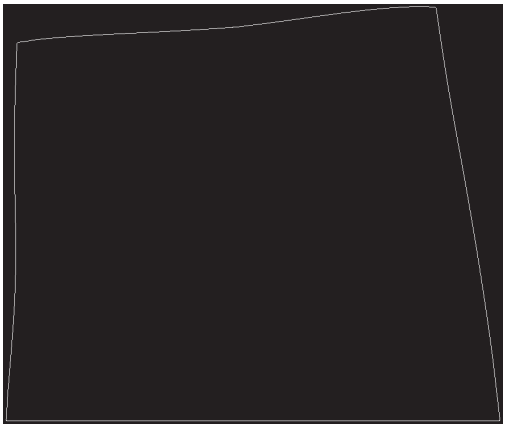

Figure 3.11: Mode shape image example after applying the Canny edge detector. Notice all of the strong edges in the image are shown.

should be evaluated. Afterwards, the strong edges of the mode shape are computed using the Canny edge detector [68]. Thse edges are represented as white (unsigned 8-bit integer value of 0) binary points. The edges of the example image is shown in Figure 3.11.

Image Contours: When using the Canny edge detector, often more than one strong edge is identified. For the purpose of this work, only the outer-most edges of the mode shape image are needed. Using the Suzuki Border Following method [69], the external contours of the edges can be calculated, thereby leaving only the outer most contours as shown in Figure 3.12. These contours represent a simple curve joining all continuous points along a boundary that have similar intensity or colors. In this work, this curve is represented as a vector of points joining all the continuous

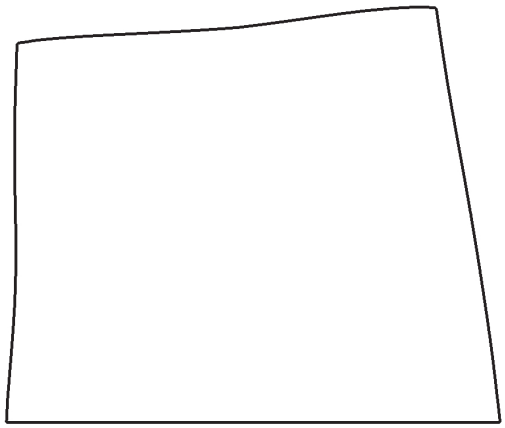

Figure 3.12: Mode shape image after calculating the outer most contours. The strongest contour is shown here and is represented as an array of black pixel values.

points. If there are multiple continuous lines found by the Suzuki method, the external contour is assume to be the one that represents the mode shape's outer-most edge.

Calculating and Sorting Corners: With the exterior contour found previously, the corners of the mode shape are found using the Shi-Tomasi corner detector [63] as implemented in OpenCV. This algorithm returns four (x,y) coordinates, which represent the pixel locations of the corners of the mode shape. Given a maximum of four corners per image, the next step is to label the corners as top-left, top-right, bottom-left, and bottom-right. This is done by sorting the (x,y) pixel coordinates, first sorting them by x-value, followed by the y-value as a secondary sort. After sorting, comparing the first two points' y-values determine which point will be the top-left or bottom-left point on the quadrilateral. The same comparison is then made on the last two points (top-right, bottom-right) in the sorted list. In addition to the corner points, a center point is determined using the previously calculated corner data. Calculating center by using the moments of the contours points is a simple way to find the center of the mode shape image.

In early testing using this method, the corners of the image were often mislabeled. It was found that some of the sharp edges and image artifacts were often mistaken for corner points. Tests should be done to determine the needed hyper-parameters for blurring and corner detection to remove possible disjointed edges and residue artifacts from image generation.

Creating Equally Spaced Points: Given four corners and the (x,y) locations of the outer edge points of the mode shape, the next step is to identify an equally spaced series of points on the outer edge of the mode shape. Prior to creating equally spaced points, the number of segments

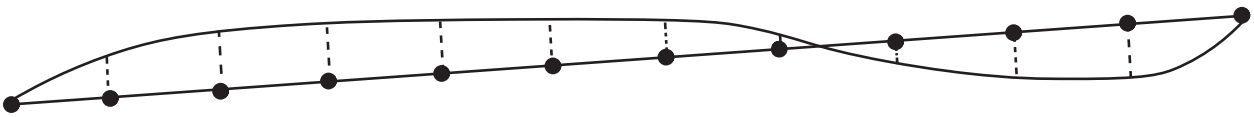

Figure 3.13: Graphical representation of calculating equally spaced points on an edge of a mode shape image. First, two adjacent corner points are used to calculate a linear equation. In this case, it is the top-left and top right points. A linear equation is calculated and D+1 equally spaced points are selected from the line (including the end points). This is represented by the solid straight line and the points on that line. The edge of the mode shape image is represented by the curved line with possible convex and concave curvature. This curved line is made of discrete points that represent pixels of the mode shape from the calculated contours. Using the guide line's equally spaced points, true mode edge image points are located and selected as corresponding points. This can be seen in the dashed lines.

needs to be chosen. As seen in Figure 3.9, there are 11 segments that divide the top edge in the example. The number of segments, D, was experimentally chosen (described later). To create an edge with equally spaced points, a guide line is first created. This is done by calculating a linear equation of pixel locations with two adjacent corner points. Given the linear equation and D, linear interpolation is used to find D equally spaced segments (or D+1 equally spaced points) between the two corner points on the guide line, demonstrated in Figure 3.13.

The guide line's equally spaced points are then used to find the nearest point from the mode shape contour points list. The edge list contains locations of pixels that were labeled as strong contour edges as found previously. Using the contour list, the pixel locations are placed into a k-d tree [70] to enable a fast way of finding nearest neighbors. Afterwards, each point in the guide line is input into the k-d tree to find the closest contour point. A true mode edge (x, y) location is returned and stored for that guide point. This is done for each guide line point. Finally, the two corner points are then included in the true edge point list. An edge of equally spaced points on the mode shape edge are now associated with the guide line. This is then repeated for the other three edges.

Making Triangles: After finding the equally spaced edge points and the center point of the mode shape, triangles can now be created. This is done by creating triangles for every two adjacent edge points (inclduing the corners) and the center point for each edge. Figure 3.9 details this operation. Triangle 1's vertices are made from the center point along with the top-left point,

and the next point. Triangle 2 is made from triangle 1's right point, the 3rd guide line point and the centroid. This continues till D triangles are created.

Mapping a Mode Shape Image to a Template Square: The template square is used to make sure that each mode shape image is a uniform shape to allow for better recognition by removing white space and extreme concave and convex edges. In this work, the template square is an 100×100 pixel image, with D equally spaced regions on each edge. This can be seen in Figure 3.9, on the right side. The same algorithms described previously to create triangles for the mode image were also applied to the template square.

After triangles are create in the mode shape image and the template square, the mode shape image triangles are mapped to the template square. Given six points, (three pixel locations from a slice from the mode shape image and the three corresponding pixel locations in the template square), an affine transformation [66] is used to warp the mode shape triangle slice to its corresponding slice on the template square. The general equation form of the affine transformation is given by:

$$\begin{bmatrix} t_1 & t_2 & t_3 \\ t_4 & t_5 & t_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.2}$$

or

$$\mathbf{TA} = \mathbf{X} \tag{3.3}$$

where $\mathbf{T}$ is the transformation matrix which define the scale, rotation, translation, and shear coefficients, $\mathbf{A}$ are pixels from the original triangle, and $\mathbf{X}$ are the corresponding pixel coordinates in the transformed triangle. Triangle 1 (tri1) is made of three points, where each point (x,y) represents a pixel location. For the original image, tri1 contains: (a_1,a_2), (b_1,b_2), and, (c_1,c_2). Similarly, triangle 2 (tri2) contains: (x_1,x_2), (y_1,y_2), and, (z_1,z_2).

Given two triangle point sets, tri1 and tri2, the first step is to calculate bounding rectangles for each triangle and label them r1 and r2. Next, each of the triangle points are offset by their respective top left corner. This is done by subtracting the x and y coordinates of the top left

corner from the other two vertices in the triangle set. These offset points are called tri_cropped and tri2_cropped. This procedure offsets the new triangle points to be zeroed to the top left corner. After this, r1 is used to crop the main mode shape image and is labeled img1cropped. Then, the affine transformation coefficients, **T**, are calculated using Equation 3.2. This function calculates the coefficients that govern rotation, scale, shear, and translation from one image to another, given two sets of three points, tri1_cropped and tri2_cropped. Affine warping maintains that parallel lines from one image will continue to be parallel in the warped image. After the coefficients are found, the warp is performed on the cropped image (i.e. img1cropped) using Equation 3.2 with the previously found coefficients. Now, the mode shape image is affine warped, but the triangle slice is still bound by a rectangle. A mask is used to black out all of the pixels outside of the triangle slice. After that, the mode shape image triangle slice has been warped to the template square. This algorithm is repeated for each slice on all four sides to create a complete warp.

Post-processing and Cleanup: After warping, there are often some artifacts that are left over on the outer edges of the square. Figure 3.14 demonstrates the random pixel values that occur on the outside edge of the entire square mode shape image. Investigating these pixels show two possible reasons why they are present. First, the artifacts are generated with the original mode shape image. Second, there are times when a triangle slice intersects with a part of the original mode shape image because of concave or convex curves on a triangle's edge. It is convenient to disregard these pixels because the mode shape image is still the dominant shape in the square mode shapes. To remove the excess information and to create a visually appealing warped image, the edges of a final warped image are replaced with black pixels.

Choosing the value of D: In the examples shown prior, D was chosen to be 11. The value of D is chosen after evaluating two criteria: refinement needs and image curvature. Refinement needs can be thought of as meshing the image. A finer mesh, in this case more triangles, allows for more curvature to be captured during the triangulation phase. One might think that the most amount of triangulation would be best; however, tests have shown that because of the edge's curvature, there are times when multiple points on the guide line corresponds to a single point on the mode shape edge. This is because the edge data are discrete pixel locations and not a continuous function. This creates errors while creating equally spaced points step. as can be seen in Figure 3.15. In this Figure, D was set to 200. Notice the tearing and the misplaced control points for some of

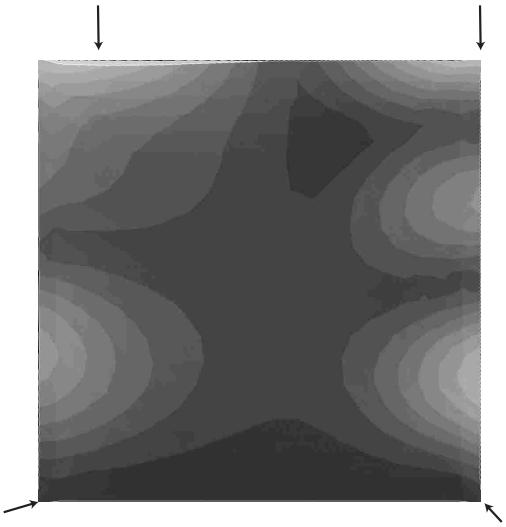

Figure 3.14: Edge artifacts after the triangle warping method. These artifacts are left over because of truncation of edges or artifacts from the original image.

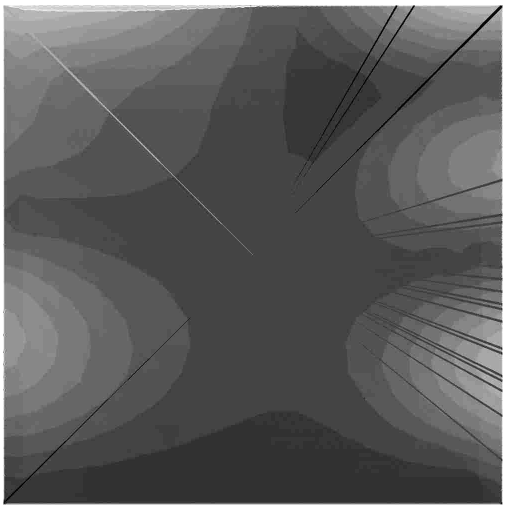

Figure 3.15: Determining the value of D depends on the refinement needs and the image curvature. This Figure shows when the value of D is too high, thereby causing triangle creation failure.

the triangles. A balance needs to be found between these two competing criteria. In this work's training and test data sets, setting D to be 20 was found to work well in balancing the two criteria for selecting D.

3.3 Principal Component Analysis

Feature extraction is performed using PCA as described previously in Section 2.6. The number of PCs extracted is set to P, a predefined value designated by the user. The number of PCs needed must be investigated by the user to maximize the classification accuracy of the learner, while minimizing the number of PCs needed to reduce the computational cost. This can be done by testing a labeled test set, or using *k*-folds cross-validation, as will be describe later in this chapter.

3.4 Training the machine learning classifier, support vector machines (SVM)

Support vector machines were described previously in Section 2.3.3. Using training data, this algorithm creates a hyper-cylinder that maximizes margin between two classes. Each labeled training image in this work is extracted by PCA to become a row vector of length P. These PCs of each image, in addition to their labels, are then inputted into the SVM learner and the classifier function is generated. There are a few hyper-parameters associated with SVM that are important to mention: kernel type, probability, and the type of multi-class classification decision function. The kernel defines the type of line that the SVM algorithm draws. As stated in Section 2.3.3, this algorithm finds a line that separates the data. However, a kernel can be used to change the function from a line to either a radial basis function, polynomial, sigmoid, or a user defined function. This kernel allows the algorithm to draw a line that can potentially fit non-linear data. The probability score was discussed when SVM's were introduced. Part of the appeal of using the probability is that it gives an estimate of how confident the classifier is at predicting the current test sample. In some SVM implementations, this can be toggled on or off, since it does have significant processing costs. Last, the type of multi-class classification method. A one-vs-one and a one-vs-all can be taken while using most implementations of SVM.

3.4.1 Cross-validation

Cross validation is a methodology that assesses and predicts how well a classifier would perform against an unknown test sets [22]. In machine learning, often k-folds cross validation is used to assess a classifier [22]. This method takes a training set and divides it into k-equally sized subsets, each with approximately the same number of samples per class for a multi-class problem.

Given k-folds, one of the folds is retained as the validation set whereas k-1 folds are used as the training set and a trained classifier is calculated. The trained classifier is now used to predict the classes of the fold that was left out as the validation set. This strategy is then repeated with the other k-1 folds, where each fold it used as the validation set while the rest are used to train a classifier. The performance of each fold is then averaged, thus allowing a good understanding of the overall training set. Though cross validation methods are not perfect because they remove a part of the training set, methods give good insight on how well a classifier might work without going through the tedious task of manually classifying the test set. Usually, the accuracy is used as the performance measure while evaluating a classifier. Accuracy is discussed later in this chapter.

3.5 Identifying Unknown Images

After training the mode shape classifier with the training set, a function is now available to classify unknown images. The unlabeled test images go through the same process described by Figure 3.1. A test set from a similar DOE is generated with unknown mode shapes. Each mode shape is preprocessed to a standard shape. Afterwards, PCA is performed on each image to extract the PCs necessary for input into the classification function. The test image's feature vector then is input into the classifier and a mode shape prediction, along with the probability score, is returned for each image. Each mode shape image is then sorted into a file structure for future evaluation.

3.6 Performance Measures

After training a machine learning classifier, the performance of the classifier is evaluated. Evaluating the training set is usually performed by the method described previously using labeled data in a cross-validation method. Measuring the performance of a test set is more difficult. Whereas the training set has known labels, the test set does not. Therefore, either a human user must manually label the test set to find the truth of every example, or the test set is simply not measured. In the case where performance can be measured, there are commonly three types of performance measures used: accuracy, precision, and recall (which will be described in the next subsection). Another tool often utilized for measuring the performance of a classifier is the confusion matrix.

A confusion matrix is often used to visualize the performance of a classifier. The name comes from how easy it is to look at the matrix and tell which classes are mislabeled as one another [22]. Each row represents the class a particular sample was labeled, and the column represents the actual label of the sample. A classifier with perfect performance would have a diagonal confusion matrix. A confusion matrix is particularly useful when there are multi-class, classification problems. A user can look at the matrix and tell which classes are being mislabeled and which class label a sample is given. In addition, a confusion matrix also facilitates the calculation of accuracy, precision, and recall. A summary of the strengths and weaknesses of these performance measures are now given.

3.6.1 Accuracy

Accuracy is defined with the following equation [22]:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.4}$$

where **TP** = True Positives, **TN** = True Negatives, **FP** = False Positives, **FN** = False Negatives. Unknown labeled images are considered in this calculation. Should an image be labeled as an unknown image that was trained on, this is a false negative. If an image is labeled as a known image, and should have been an unknown, then it is a false positive. True positive means the mode image was labeled correctly and true negative means an unknown image was labeled correctly.

Accuracy is often the first measure that is used to get a general global understanding of the training data and the test data. It gives a general understanding of how well the classifier performs using a given machine learning algorithm and training data. However, there are limitations to using accuracy alone as is often see in the accuracy paradox [71]. For example, the performance of an email spam detector is given in Table 3.1.

This sample of classified email results as shown in Table 3.1, using Equation 3.4 gives the accuracy of the of the classifier: $Accuracy = \frac{10+15}{10+15+25+100}$, which equals an accuracy of 73.3%. These results show that the classifier, though not perfect, still does well for this particular data set.

Table 3.1: Confusion matrix of an example email classifier. A classifier was trained prior to this test and the results are represented here. This data shown that the classifier does well, though not perfect.

	Classified Positive	Classified Negative
Positive Class	10 (TP)	15 (FN)
Negative Class	25 (FP)	100 (TN)

Table 3.2: Confusion matrix of an example email classifier using a naive approach without any training. All of the samples were labeled as "not spam". This table shows the aggregate truth for all of the samples.

	Classified Positive	Classified Negative
Positive Class	0 (TP)	25 (FN)
Negative Class	0 (FP)	125 (TN)

However, when looking at a naive case where every email is blatantly labeled as "not spam" the confusion matrix becomes Table 3.2.

Again using Equation 3.4 to calculate the accuracy of the classifier against the data set gives the accuracy of: $Accuracy = \frac{125}{25+125}$, which equals an accuracy of 83.3%. In this case, it was better to guess all emails were not spam than to go through the process of training a particular classifier. This is why other metrics should be used when determining the quality of a machine learning algorithm, training set and/or test set.

3.6.2 Precision and Recall

In addition to accuracy, precision and recall are other metrics often used to measure the quality of a classifier. Precision is the ratio of correct positive examples to the number of actual positive examples [22]. By the definition just given, precision is defined as

$$Precision = \frac{TP}{TP+FP} \tag{3.5}$$

where TP and FP are the numbers of true positive and false positives, respectively. This measurement ranges from $0 \leq Recall \leq 1$, where 1 is a perfect score. High precision (1.0 meaning a perfect score), represents the relevance of the classifier. It shows, of all the samples identified as class a,

how many were labeled correctly [72]. However, it does not say whether all relevant examples were found.

Recall is another performance measure that dictates how well the classifier finds all relevant classes from the test set. In contrast to precision which measures of all the classified data, how many were labeled correctly, precision takes a holistic approach and measures, of all possible samples that *were actually* class *a* in the whole test set, how many of them were labeled correctly. However, it says nothing of whether false positives were also labeled. The equation for recall is

$$Recall = \frac{TP}{TP+FN} \tag{3.6}$$

This measurement ranges from $0 \leq Recall \leq 1$, where 1 is a perfect score.

Revisiting the email filter from before, the Precision of the trained classifier is $Precision = \frac{10}{10+25}$, which equals 28.5%. The recall score is, $Recall = \frac{10}{10+15}$, which equals 40%. Comparing this to the naive classifier, the precision score is $\frac{0}{0+0}$, which gives an undefined answer, and the recall score is $\frac{0}{0+25}$, which equals zero. From the precision and recall scores shown here, both the trained classifier and the naive classifier performed poorly. Precision and recall are often used together to give more depth to the accuracy measurement. This work takes advantage of these three measurements to qualify the classifiers created to identify mode shape images.

CHAPTER 4. IMPLEMENTATION: THE EIGENBLADES CLASSIFICATION ALGORITHM

4.1 Overview

This chapter discusses the implementation of the algorithms described in Chapter 3. Code listings and pseudo code will also be given here detailing the specific algorithms that were described. In addition, external libraries referenced in Chapter 3 will be described in more detail.

4.2 Generating Mode Shape Images

This research required a large number of mode shape images to be able to have a proper training set and test set. The first outcome of this research is a Python DOE implementation that automatically generates mode shape images of turbomachinery blades. In this work, Python was used as the framework that joined programs together to randomly sample the design space and generate the solid geometry needed to produce modal images. The pseudo-code overview to generate mode shape images is seen in Figure 4.1.

4.2.1 Design of Experiments Setup

To keep track of the changes that are happening within the design of experiments, a text file, called the input file, is created for every experiment. This file governs the geometry of the blade for a particular experiment. It is used as an intermediate step that links the random sampling and the NX program. For example, Figure 4.2 shows an input file for a particular blade. The first seven lines define the blade's save number, number of sections, and parameters that are generated from the random sampling. These five values define the nominal value of the blade. Each subsection (lines 8-13, 14-20, etc.) then describes the parameters of each section of the blade. First, the airfoil is defined. In this example, the NACA-M22 airfoil. Afterwards, each value defines the

```
Set nominal, minimum and, maximum values for blade parameters
while experiment-index ≤ number-of-experiments do
  Set Latin-hypercube samples for experiment-index
  while run-index ≤ number-of-runs do
    Get Latin-hypercube sample of samples for run
    Write input file
    Call NX to create geometry
    Call ANSYS to perform modal analysis and generate pictures
    Store images
  end while
end while
```

Figure 4.1: Design of experiments to create mode shape images.

percentage from the nominal that is changed for that section. For example, line 9 indicates that the base section's chord length is the nominal value, whereas every other parameter is set to 0%, i.e., unchanged. The next section (lines 14-19) shows that the chord length is 95% of the nominal and each of the other parameters as 33% of the nominal.

4.2.2 Blade Geometry

The structural models used for this work were generic aerospace application blade shapes. The airfoil shape selected was the NACA-M22 blade airfoil which can be easily accessed on publicly available websites. This shape does not have any significance to the method, rather it is given for the reader's information. Any airfoil shape can be used. This blade shape was subject to a DOE that changed the values of the parameters including cross-sectional areas, chord, twist of the total blade as measured from the base to tip, chord-offset, and total length of the blade.

For the experiment, design parameter range was set, as shown in Table 4.1. The blade's variables as discussed previously in Chapter 3 are shown in this table. A Latin hypercube [73] sampling was then used to vary the design variables of the base values within the range set by Table 4.1 (discussed in the next section). These values are the minimum, maximum, and nominal values of the five variables that define the design space used in the DOE.

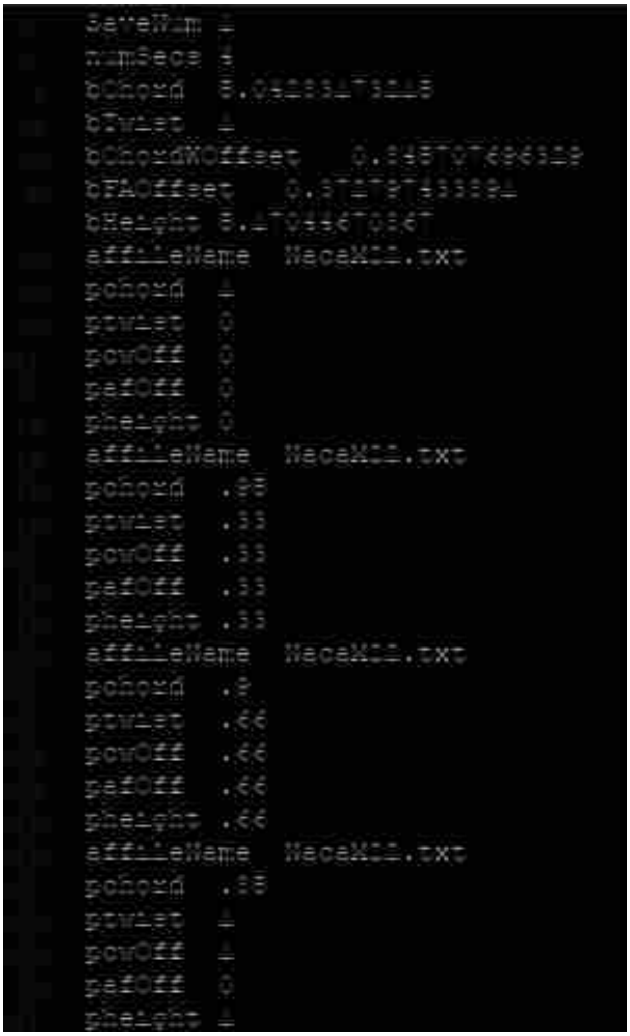

Figure 4.2: Input file that governs the blade geometry. This file is used as an intermediate step between Latin-hypercube sampling and creating NX geometry.

Table 4.1: Blade design range for each parameter. Base design parameters are set to the nominal (Nom) values. As the DOE is run, the values will change according to a Latin-hypercube between the minimum (Min) and maximum (Max) values.

	Chord	Twist	CWOffset	FAOffset	Height
Min	2.750	0.750	0.275	0.0275	2.250
Nom	5.500	1.500	0.550	0.550	4.500
Max	8.250	2.250	0.825	0.825	6.750

```
import pyDOE

def getDoESet(n=5, samples=10):
    # n: an integer that designates the number of factors (required)
    # samples: an integer that designates the number of sample
    points to # generate for each factor (default: n)

    design = pyDOE.lhs(n, samples)  # constructor for pyDOE
    means = [5.500, 1.500, 0.550, 0.550, 4.500]  # parameter nominal values
    sdtvs = [0.5, 0.5, 0.5, 0.5, 0.5]  # change from the nominal

    for index in xrange(len(means)):
        design[:, index] = norm(loc=means[index], ...
        scale=sdtvs[index]).ppf(design[:, index])

    return design
```

Figure 4.3: Implementation of pyDOE. This code creates the random sampling for one experiment.

4.2.3 Latin-Hypercube Sampling

Latin-hypercube sampling was implemented in Python using the `PyDOE` external library. This library is designed to help scientists, engineers, statisticians construct experimental designs. It contains functions that allows the user to perform full factorial designs and randomized designs like Latin-hypercube. As described in Chapter 3, a Latin-hypercube sampling seeks to explore the full design space while minimizing the number of samples needed to get statistically near random samples. Given the nominal values and range in Table 4.1, the Latin-hypercube random sampling is performed by the Python function shown in Figure 4.3.

The hyper-parameters for the pyDOE Latin-hypercube function, `pyDOE.lhs` are: n, an integer that designates the number of factors (parameters), samples, an integer that designates the number of sample points for each parameter. In this work, n is set to 5 because there are 5 parameters that govern the blade geometry and samples is set to 10 or 20, depending on the number of mode shapes generated per experiment. This creates a randomly sampled, equally spaced data matrix of size 5×10 (for a 10 mode run) that defines the parameters for one experiment within a DOE.

```
Read input file
inputparams ← numsections, chord, twist, bChordOffset, Forcaft-offset, height
Set parameters for Blade.prt
Save Blade.prt file
```

Figure 4.4: NX API to create solid geometry

The design space is created by the `means` and `sdtvs`, where `means` are the nominal values for each parameter and the `sdtvs`, designate the number of standard deviations the samples can vary from the mean. The pyDOE returns values between 0 and 1, so the for loop that finishes the function scales each parameter to the mean.

4.2.4 NX Geometry

Throughout the DOE, as new blade parameters were generated by PyDOE, they would be input into Siemens NX, a 3D CAD modeling software, by the input file created by the previous listing to automatically create 3D geometry that would subsequently be used for modal analysis. A script was written to automate the process of exploring geometry parameters through Latin-Hypercubes, reading in the blade parameters from an input file, and generating a blade using NX's application program interface (API), NX OpenAPI written in Java. This program was created to take the blade parameters generated from the DOE sampling and create blade geometry. The pseudo-code for the Java program is shown in Figure 4.4.

4.2.5 Modal Analysis in Mechanical ANSYS

This work uses Mechanical ANSYS to perform modal analysis. To automate the process of performing modal analysis with a DOE, a script was written for Mechanical ANSYS using its API design language called ANSYS Parametric Design Language (APDL). Images were then captured from ANSYS with 10 or 20 modes per blade. The 20 mode DOEs were used to get training images. The 10 mode DOEs were used to generate the test images set. The mode shape images were generated in gray-scale. The pseudo-code to generate these modal images is as follows:

```
Set material data
Select Element Type: Solid 285 TET
Meshing the geometry to 0.55 length size
Constrain root node elements to 0 displacement
Set number of modes to find (10 or 20)
Set up modal analysis with the Block-Lanczos solver, with expanded mode shapes
Setup grayscale controls for image capturing
Print JPEG image for each mode shape
```

Figure 4.5: Mechanical ANSYS APDL to perform modal analysis

4.3 Mode Shape Image Examples

Although, there are many possible types of modes and even combinations of modes that can be generated from Mechanical ANSYS, this research limits mode images to those that gave distinct images. Higher-order modes that contained multiple dominant mode shapes or modes that were unclear to a human expert are not considered and are beyond the scope of this research. Figure 4.6 shows the 16 distinct modes this work seeks to classify. In addition, since each mode image represents a physical displacement, the type of blade bending is listed with each mode number. The mode numbers are used to simplify the classification labels. These mode shape names and numbers are described in Table 4.2

As the DOE is executed, many blade geometries are generated. Different geometries can have the same mode shape. For example, Figure 4.7 gives several examples of the mode shape labeled as mode 2. Notice that the mode shape is not symmetric, neither are each of the individual blade's geometry the same, but a user can tell they are all the same mode. In some cases, the blade itself is not a perfect quadrilateral and the size and orientation of these blades differ from each other. Larger perturbations and even non-quadrilateral blades were considered but are not shown. In addition, because of these differences, two blade shapes can have different unique mode shapes. For example, one blade can have mode shapes 1–10, while another may have mode shapes 1–5, 8–12. Thus, 16 separate modes were trained, since there is no guarantee which mode shapes will appear in each run within a DOE using ANSYS APDL.

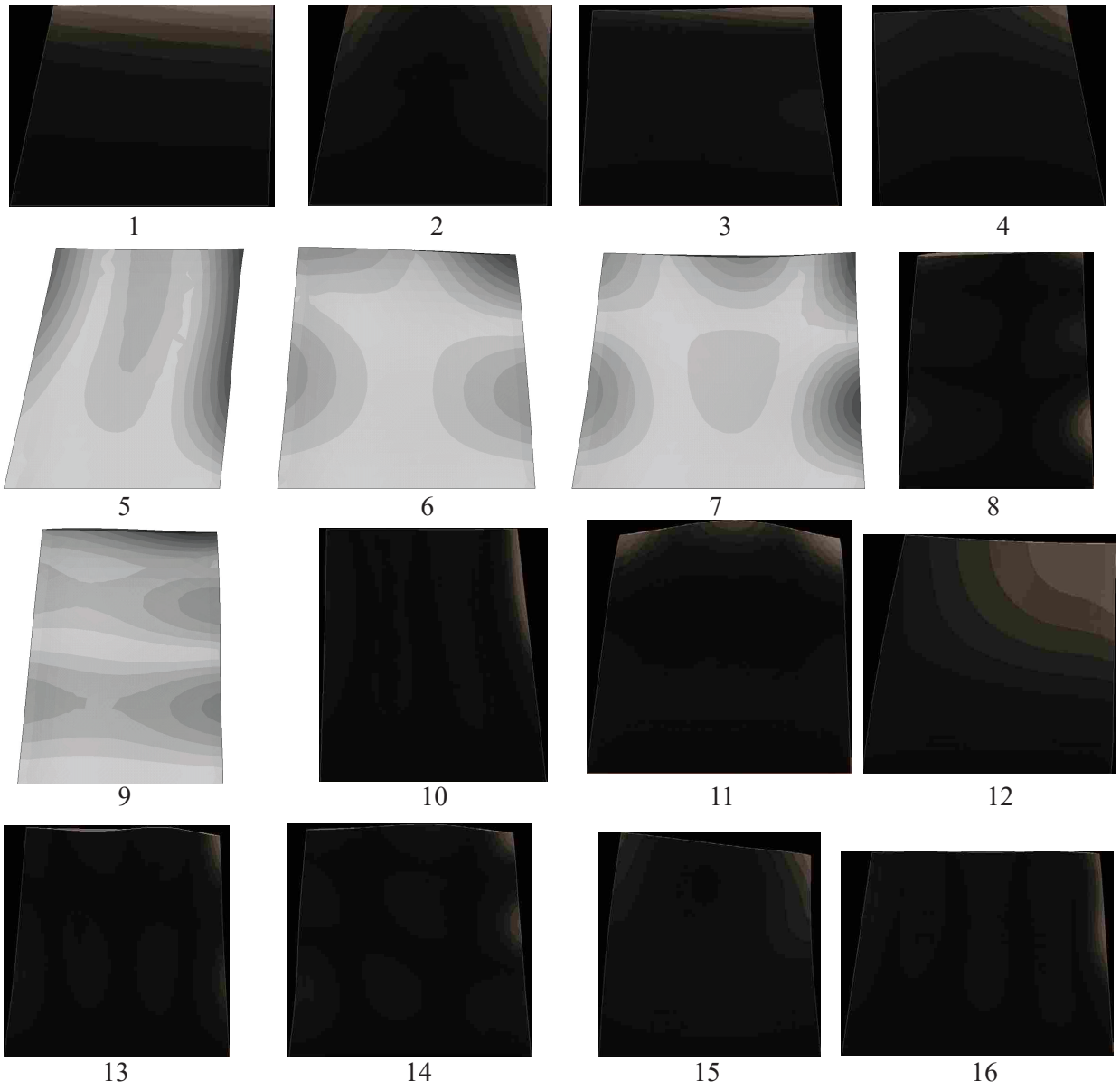

Figure 4.6: Dominant mode shapes up to the 16th order. These mode shapes show the possible 2D projections of deflections that can be caused by dynamic cyclic loading on a turbomachinery blade.

Table 4.2: Vibrational bending mode names for turbomachinery blades.

Mode Number	Number of images
Mode 1	First Bending
Mode 2	First Torsion
Mode 3	Leading and Trailing Edge Torsion
Mode 4	Tip First Torsion
Mode 5	First Chordwise Bending
Mode 6	Second Torsion
Mode 7	Second Chordwise - Second Bending
Mode 8	Third Torsion
Mode 10	Second Chordwise Bending
Mode 11	Second Chordwise - Second Torsion
Mode 12	Complex Trailing Edge Bending
Mode 13	Third Chordwise - Second Bending
Mode 14	Second Chordwise - Third Bending
Mode 15	Complex Leading Edge Tip
Mode 16	Third Chordwise Bending

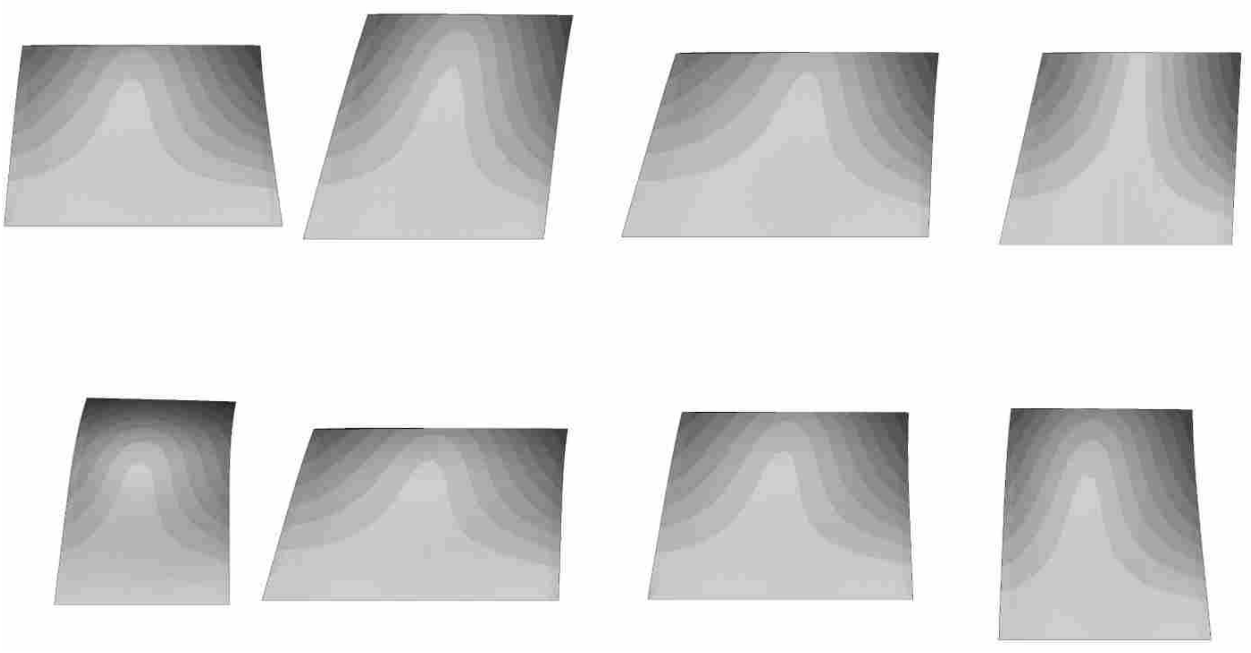

Figure 4.7: Images of mode 2 with varying geometry. Notice how the mode shape differs slightly for different blade configurations, but is the same mode throughout the geometry changes.

Table 4.3: Number of training images per distinct mode shape. In general, each mode was given as many as possible training images to cover the training space as much as possible.

Mode Number	Number of images
Mode 1	163
Mode 2	604
Mode 3	627
Mode 4	686
Mode 5	455
Mode 6	553
Mode 7	634
Mode 8	126
Mode 10	629
Mode 11	20
Mode 12	115
Mode 13	173
Mode 14	69
Mode 15	118
Mode 16	173

4.3.1 Generated Data

Ten different DOEs were run to generate thousands of images. One of the DOEs was chosen to be part of the test set and the rest were used as training images. This work distinguished 16 different modes with 5,293 training images in total that were spread through 16 mode shapes. Each training mode image was selected to have a substantial amount of visual variance from every other training image within a particular mode. Table 4.3 details the number of training images per mode.

The reason for the disparity between the number of training mode shape images between modes is two-fold. First, some modes occurred less often during image generation. Notice that mode 11 only has twenty images. Of the DOEs that were performed, this mode only appeared 11 times in the 9 training DOE image sets. It seems that mode 11 is an uncommon mode shape, therefore, the test set should not contain very much of them either. In any case, mode 11 should be seen as a class that carries less weight in evaluating the Eigenblades classification method. Second, in general, the goal behind the number of training images was to have as many as possible. The notable exception is mode 1. Mode 1 is the most common mode shape and because of its simplicity, experimental results showed that this mode did not need many training images. Therefore, to

Table 4.4: Number of images per distinct mode shape in the test set.

Mode Number	Number of images
Mode 1	273
Mode 2	228
Mode 3	200
Mode 4	191
Mode 5	208
Mode 6	195
Mode 7	197
Mode 8	113
Mode 10	42
Mode 11	5
Mode 12	45
Mode 13	1

reduce computation time, mode shape pictures that seemed most dissimilar were chosen to train mode shape 1.

The test set contains 1,922 images. This image set only contains images that were deemed as a distinct mode shape as described earlier. The test set was generated by having a human expert go through the DOE run that was set aside as the test data. The expert went through each image to determine if the image was a distinct mode shape, and then, labeling it if it was. No image in this set is used to train the classifier. This test set is used to determine the performance of the classifier by using the true labels given by the human user, and comparing it to the labels given by the classifier. The number of images per mode in the test set is given in Table 4.4. The performance measures described previously are used to compare the true labels against the classifier's labels.

Notice there were zero images for modes 14, 15 and 16. The DOE that created these images did not produce these modes. Like the training images, these number of test images were limited to how common mode shapes were during modal analysis.

4.4 Preprocessing

This section outlines the libraries used and details the scripts that was written to perform the bilinear and triangle interpolate warping algorithms. In general, the Python implementation

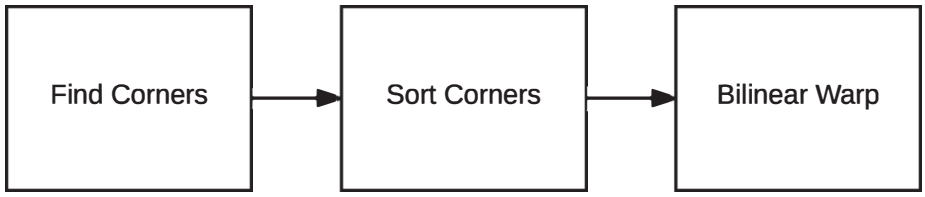

Figure 4.8: High-level preprocessing algorithm for bilinear image warping. The purpose of preprocessing each image is to scale each image into a standard shape and to remove as much white space as possible so only the blade is used for feature extraction.

```
import numpy as np
import cv2

def cornerdetection(self, image):
    # make sure image is a grayscale image for corner detection
    gray = np.float32(image)
    # initialize an list to store corners
    actualcorners = []
    # find corners
    corners = cv2.goodFeaturesToTrack(gray, 4, 0.01, 10)
    corners = np.int0(corners)
    # extract corner pixel locations from other data
    for corner in corners:
        point = corner[0].tolist()
        point = tuple((point[0], point[1]))
        actualcorners.append(point)
    return actualcorners
```

Figure 4.9: Finding the corners for the bilinear warp

of OpenCV was used for all algorithms [65]. All images were read as single-channel, grayscale images prior to image processing.

4.4.1 Bilinear Warping

The bilinear warping follows the high-level preprocessing method presented in Chapter 3, reproduced in Figure 4.8. First, the corners of a mode shape image were found using the OpenCV implementation of the Shi-Tomasi corner detector [63], and was implemented as seen in Figure 4.9.

```
import numpy as np
import cv2

def cornerdetection(self, image):
    # make sure image is a grayscale image for corner detection
    gray = np.float32(image)
    # initialize an list to store corners
    actualcorners = []
    # find corners
    corners = cv2.goodFeaturesToTrack(gray, 4, 0.01, 10)
    corners = np.int0(corners)
    # extract corner pixel locations from other data
    for corner in corners:
        point = corner[0].tolist()
        point = tuple((point[0], point[1]))
        actualcorners.append(point)
    return actualcorners
```

Figure 4.10: Bilinear warping using the PIL Image.transform class function

The four corners are then sorted into a list where the *bottom-left* corner is first, and followed by *top-left*, *top-right* and, *bottom-right*. These corners are necessary to use the bilinear warp, as described in Equation 3.1. The second set of coordinates used are a standard square where its values starting at the bottom left and continuing clockwise are: $(0,0)$, $(0,100)$, $(100,100)$ and, $(100,0)$. These two sets of corners form the system of linear equations that can now solve for a_0, a_1, a_2, a_3, b_0, b_1, b_2, and b_3. Given these coefficients, each pixel in the mode image can now be warped to the template square using the bilinear warping algorithm which was implemented shown in Figure 4.10.

This work utilized the bilinear transform implemented in the Python Image Library (PIL). Given the size of the final image (100×100), the type of transform, the previously calculated coefficients, and the type of interpolation used, the transformed image is stored in `img`.

```
import cv2

image = cv2.GaussianBlur(image, (5, 5), 0)
# blurs within a size 5,5 box 1 standard deviation
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
# creates a 5x5 structural element
image = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
# operates on the blur
edges = cv2.Canny(image, minVal=100, maxVal=500,
edges=None, apertureSize=3, L2gradient=True)
```

Figure 4.11: Implementation of the Gaussian blur and morphological open operators

4.4.2 Triangle Interpolate Warping

Given a grayscale image, the triangle interpolate warping follows the algorithm given in Figure 3.8. This section describes the implementation of this algorithm, specifically the functions used to perform each step within the algorithm.

Processing and edge finding: After opening the image within the script as a single-channel, grayscale image, the first action is to clean up the image and smooth the edges by using a Gaussian blur and morphological open operator. Afterwards, the edges are founding using the Canny edge detector. These OpenCV functions are implemented as seen in Figure 4.11.

The `GaussianBlur` function takes the grayscale image, a $(5,5)$ kernel and blurs everything within one standard deviation of the mean image. The `getStructuringElement` function creates a $(5,5)$ kernel that is used to the `morphologyEx` operation. The kernel governs the step size of the morphological open operator on the given image. The `Canny` edge detector function takes the blurred image and finds the edges given the minimum and maximum threshold values, aperture size, and the gradient used. The option `L2gradient` is set to true to allow for the true gradient of the image to be calculated, thus increasing the accuracy of the edge finding. The Canny edge detector works by finding large gradients in the image. These large gradients are assumed edges if they have a long chain of connected points that lie above a certain threshold. This threshold is governed by the minimum and maximum values. Any edges with intensity gradient more than `maxVal` are edges and those below `minVal` are non-edges, and therefore are discarded. Those who

```
import cv2

contour_image, contours, hierarchy =
cv2.findContours(edges,mode=cv2.RETR_EXTERNAL,method=cv2.CHAIN_APPROX_NONE)

blankimage = 255 * np.ones((shape), dtype=image.dtype)
cv2.drawContours(blankimage, contours[-1], -1, (0, 255, 0), 3)
```

Figure 4.12: Finding and drawing the external contour

lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded [65]. The `aperatureSize` argument is used with the Sobel operator which finds strong edges within an image [74].

Finding Contours: Given an image that defines the edges of a mode shape, the contours are found to facilitates the next steps in finding the corners, centroid, and connected lines. The contours in OpenCV are found using the code seen in Figure 4.12. The `findContours` function takes the edges and then finds only the external connected edges in the image. Afterwards, the contours are drawn on a blank image so they can be used in their own independent calculations.

Finding and Sorting Corners: With the external contours calculated, the corners of the mode shape can be computed and sorted. In addition, the centroid can be found using Figure 4.13. The corners are calculated using the class function, `cornerdetection`, which takes the contour image and finds the corners by location edges with large changes. A maximum of 4 corners are found. Afterwards, the corners are labeled as top-left, bottom-left, bottom-right, and top-right. The center of the mode shape is then found by averaging the x and y pixel locations of the four corners.

Creating Equally Spaced Points: A guide line is calculated to create equally spaced points that will define the triangles on the mode image. This was accomplished by using a function that found a line between two corner points and then finding the equally spaced, intermediate point in that line. This was done with the function in Figure 4.14.

Given two corner points and the number of equally spaced points desired, this function returns the (x, y) location of the desired points. This is repeated for each pair of adjacent corner points to create 4 guidelines. Afterwards, this guideline is used to find the actual points on the contour, so

```
import cv2

corners = self.cornerdetection(blankimage)  # calculate the corners
sorted_corners = sorted(corners, key=lambda x: (x[0], x[1]))
# sorts the values first by row then column
if sorted_corners[0][1] > sorted_corners[1][1]:
    top_left = list(sorted_corners[1])
    bottom_left = list(sorted_corners[0])
else:
    top_left = list(sorted_corners[0])
    bottom_left = list(sorted_corners[1])
# last two points are the right column
if sorted_corners[2][1] > sorted_corners[3][1]:
    top_right = list(sorted_corners[3])
    bottom_right = list(sorted_corners[2])
else:
    top_right = list(sorted_corners[2])
    bottom_right = list(sorted_corners[3])

def cornerdetection(self, image):
    gray = np.float32(image)
    actualcorners = []
    corners = cv2.goodFeaturesToTrack(gray, maxCorners=4,
    qualityLevel=.1, minDistance=20)
    corners = np.int0(corners)
    for corner in corners:
        point = corner[0].tolist()
        point = tuple((point[0], point[1]))
        actualcorners.append(point)
    return actualcorners
```

Figure 4.13: Calculating and sorting the corners and the centroid

the points of the triangle line on the edge of the mode shape. This is done first by creating a `KDTree` which facilitates fast nearest-neighbor searching, and then using the `getTriangleNodeList` function seen in Figure 4.15. Every point on the guide line is input into the contour point's KDTree and then the nearest neighbor to each guide point is calculated on the mode image.

Making Triangles: Given equally spaced points along the edge of the mode image and the center point of the image, triangles are created using the Figure 4.16. For every edge, each triangle

```
import numpy as np

def intermediates(self, p1, p2, nb_points):
    """Return a list of nb_points equally spaced points
    between p1 and p2"""
    # If we have 8 intermediate points, we have 8+1=9 spaces
    # between p1 and p2
    x_spacing = (p2[0] - p1[0]) / float(nb_points + 1)
    y_spacing = (p2[1] - p1[1]) / float(nb_points + 1)

    return [[p1[0] + i * x_spacing, p1[1] + i * y_spacing]
            for i in range(1, nb_points + 1)]
```

Figure 4.14: Calculating equally spaced points on a guideline

```
from scipy.spatial import KDTree
impory numpy as np

def getTriangleNodesList(self, edges, tree):
    # edges -> the guidelines edges
    # tree -> the contour points in a KDTree
    triangleNodeList = []  # initialize the actual triangle Node as a list
    for i in range(len(edges)):  # for every point in the guideline
        distance, location = tree.query((edges[i]))

        # locate the closest actual point on the mode image
        xpt = int(tree.data[location][0])  # store the x pixel location
        ypt = int(tree.data[location][1])  # store the y pixel location
        top_point = [xpt, ypt]
        triangleNodeList.append(top_point)

        # do this for every point in the guideline
    return triangleNodeList
```

Figure 4.15: Calculating triangle edge points from guideline

```
def triangle_maker(self, edge, center_point):
    # clockwise rotation always
    triangles = []
    for point_index in range(len(edge) - 1):
        triangle = (center_point, edge[point_index], edge[point_index + 1])
        triangles.append(triangle)
    return triangles
```

Figure 4.16: Creating triangles

is created by using two adjacent points and the center point to make a list of triangle node locations. These locations are used for the next step of mapping the mode image to a standard square.

Mapping from the Mode Shape Image to the Template Square: First the template square is created using the same algorithms described previously. Given that a square has four known corner points, these points can be used to create a standard guideline, whose equally spaced points are the actual points within the template square. Each edge of the square then has triangles created by using the center point and the code in Figure 4.16.

Now, given a triangle set in the mode image and a corresponding triangle set in the template square, an affine warp is used to transform the grayscale pixel values from the mode image. The code for this action is too long to post. See Appendix B to see the full implementation.

Post-processing and Cleanup: With the warping complete, interpolation of a continuous image often create discontinuities on the edges. These edges can either be left as is, or given a constant value like the median grayscale value of 127 or black. To do this, Figure 4.17 is used.

4.5 Principal Component Analysis

The main PCA interface was programmed using the `sklearn` library. To perform PCA, first the training set is preprocessed, and then each image is flattened to a vector. A matrix is then created by stacking each image vector to create the training matrix, where each column is a flattened image. An associated vector is also created which contains the corresponding labels for the image matrix. Afterwards, PCA is performed on the training set using the code shown in Figure 4.18.

```
import cv2
import numpy as np

self.final_image[:, 0] = 0
self.final_image[:, -1] = 0
self.final_image[0, :] = 0
self.final_image[-1, :] = 0
```

Figure 4.17: Creating constant values on the edges

```
from sklearn.decomposition import PCA

def convert_training_data(self):
    pca = PCA(n_components=self.num_pca_components, svd_solver='randomized',
              whiten=True).fit(self.training_data)
    self.eigenfaces = pca.components_
    eigenfaces =
    self.eigenfaces.reshape((self.num_pca_components, self.h, self.w))
    X_train_pca = pca.transform(self.training_data)
    self.transformed_data = X_train_pca
    return pca, eigenfaces, X_train_pca
```

Figure 4.18: Principal Component Analysis on the training set

Where `n_components` is the number of PCs to be extracted, and the solver is selected to be `randomized`. This parameter describes the algorithm used to perform single value decomposition (SVD) which solves the eigen-decomposition problem. The randomize parameter uses the method described by Halko et al. [75]. The training set is then fit to the PCA model using the `fit` function with the `training_data` as the input. The eigenvectors are then stored in the variable called `eigenfaces`. The function `pca.transform` performs Equation 2.14 on the training set, which creates a matrix of size $n \times P$, where n is the number of training images and P, the number of principal components extracted. The matrix now contains the feature vectors for each image within the training data.

4.6 Training the machine learning classifier, support vector machines (SVM)

The images were then trained on a SVM classifier found in `scikit`, a machine learning library for Python. SVM was chosen after running cross-validation experiments on other machine learning algorithms including *k*-nearest neighbor (kNN) and ID3's decision trees [59]. The 5-folds cross-validation accuracy for a variety of hyper-parameters and different machine learning algorithms was used to choose the best machine learning algorithm. These tests were not exhaustive, but showed that SVM gives the best generalization for the training set. A one-vs-one approach is taken, using the radial basis function (RBF) kernel. This work tested other kernels available in Python's `sklearn` machine learning library for SVM. The linear, polynomial of degrees 2, 3, and, 5 kernels were tested. Of these different kernels, RBF performed the best in accuracy, precision, and recall. In each case, 20 principal components were extracted as a baseline for each test.

This work uses the `sklearn` implementation of SVM as shown in Figure 4.19. The `kernel` determines which kernel is used as the separation line, `degree` represents the degree of polynomial used if the kernel is selected to be "poly", `decision function` represents whether it is a one-vs-one (ovo) or one-vs-all (ova) on multiclass problems, and the `probability` hyper-parameter decides whether the probability score for the classifiers are calculated. The SVM classifier is initialized using the `SVM` constructor with the previously discussed hyper-parameters. Afterwards, given the preprocessed training data and the label vector, the classifier is trained using `clf.fit(self.training_data, self.labels)`. The probability score is calculated using `clf.predict` function, where the input is the PCA score of the test mode image.

4.7 Identifying Unknown Images

After training the SVM classifier, a test sample can now be classified. Each test image goes through the same preprocesing and feature extraction as the PCA. The test image is preprocessing using bilinear or triangle warp. Afterwards, the PC-scores of the test image is calculated using `image_PC_scores = PCA.transform(test_image)`, where the test image is also a flattened image. To classify the test image, the PC-scores used as inputs to, `clf.predict(test_data)`. This returns the label of the test image that the SVM classifier predicts. Each test image was run through the classifier to produce a prediction and probability score. If the probability score

```
from sklearn.svm import SVC

self.clf =
SVC(kernel='rbf', degree=2, decision_function_shape='ovo', probability=True)

def train_model(self, training_data, labels):
    self.training_data = training_data
    self.labels = labels

    self.clf.fit(self.training_data, self.labels)
```

Figure 4.19: Training the SVM Classifier

was less than 0.50, then the mode shape images was automatically labeled as “unknown” mode. Conversely, if the probability score is greater than 0.50, then the label given during classification is kept and sorted accordingly.

4.8 Performance measures

As previously discussed, the test set used in this work was labeled prior to testing. This allows for true measurements to be made on the classifier with a real test set. After labeling each test image with the classifier, the test label and the true label for each test image is then compared. A comma separated value (CSV) file is then created to facilitate easy calculation for the other performance measures.

4.8.1 Accuracy, Precision and Recall

Each performance measure then uses the confusion matrix to calculate the accuracy, precision, and recall for each mode shape and globally using the equation described in Chapter 3. This was done in excel using the CSV file generated previously.

CHAPTER 5. RESULTS AND DISCUSSION

This chapter discusses the results of a series of experiments used to test and validate the Eigenblades mode shape identifier. First, using the bilinear warping algorithm and performing classification on the test images and second, performing the same experiment but with the triangle interpolate warping method. The results presented here will then be compared to prior work. In addition, limitations are discussed in relation to future work that could be done to improve this process and mode shape identification in general. Accuracy, precision, recall, and the computation time will be evaluated for the bilinear and triangle interpolate algorithms. When comparing to prior work, accuracy and computational time is used to evaluate the studied works.

5.1 Determining the Number of Principal Components (PCs)

A question early on during this research considered the optimal value of P. *P* defines how many PCs should be calculated for the mode shape images during shape decomposition. To test this, an experiment was performed to determine the number of PCs to use depending on which preprocessing algorithm was implemented. For each of the preprocessing algorithms, it was found that SVM with a radial basis function (RBF) kernel gave the best results. The probability score was set to 0.50 for each experiment. Figure 5.1 shows the mode image that was used to determine the number of PCs needed for both preprocessing methods during image reconstruction. This mode shape was used as the main example because it had the complexity in both the mode features and the edges on the image.

5.1.1 Bilinear Warping PCA

A comparison of the number of PCs versus the accuracy for the test set is given in Figure 5.2. Notice a significant improvement in accuracy from five to twenty PCs used in a feature vector. After approximately 20 PCs, the increase in accuracy levels off. This becomes apparent

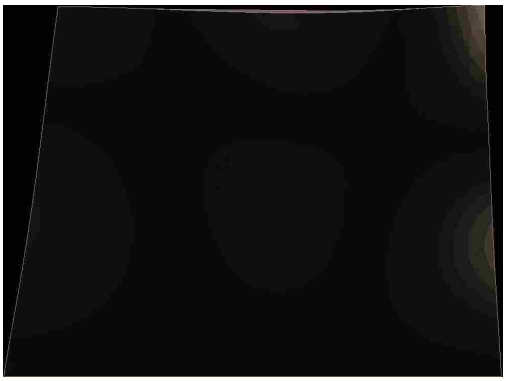

Figure 5.1: Mode shape image prior to preprocessing and PCA feature extraction. Notice the level of detail on the full mode shape prior to feature extraction through PCA. A human can easily tell its vibrational mode shape.

in Figure 5.2 where the performance measures, accuracy, precision, and recall, are shown. Notice how the accuracy continues to increase until about 20 PCs. During this increase, the precision score does not show significant changes. However, above 20 PCs, the accuracy and the recall score begin to decrease. This could be due to over-fitting the data or fitting the error/white space that is left over from the bilinear warping.

Therefore, $P = 20$ was chosen as a good number for the next experiments because it gave a fair image reconstruction with the subset of eigenvectors. For example, given the mode shape image shown in Figure 5.1, the image reconstructed from varying numbers of principal components shows the appearance variations in Figure 5.3. Since the main purpose of this work was to create an automated mode shape identification application, it would be useful to determine if the first 20 PCs would return an image that a human user can classify. Figure 5.3 shows a set of mode images reconstructed from the principal components of the image shown in Figure 5.1. Notice that the first images generated with 10 PCs or less do not reconstruct the mode image shown in Figure 5.1 very well. Only after about 15 PCs does the mode become clean enough for a human to confidently identify.

5.1.2 Triangle Interpolate Warping

Figure 5.4 graphically demonstrates a range for the variable P and the classification performance on the test set. It can be clearly seen that above 20 PCs, the accuracy, precision, and recall

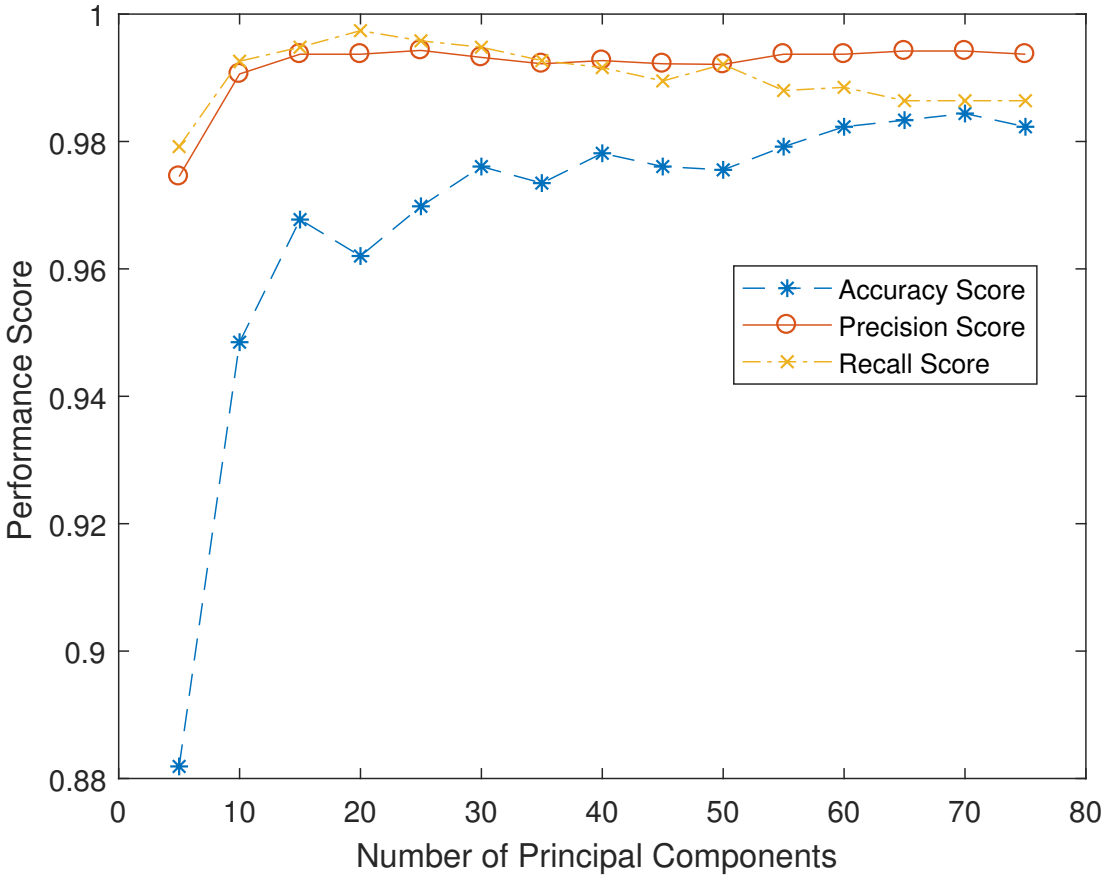

Figure 5.2: Average accuracy, precision, and recall scores of the bilinear Eigenblades methods on the test set. Above 20 principal components, there is little gain in accuracy. However, even with the small gain in accuracy, the precision, and recall scores lower significantly. This could be due to over-fit of the data above 20 PCs.

tends to level off and stops increasing at a significant rate. Unlike the bilinear warp, however, these performance measures do not decrease with more PC scores. This could be because the triangle warp method does not exacerbate the white-space errors like the bilinear warp method. It can be concluded that a P value above 20 will not give significant increase in performance for the triangle interpolate Eigenblades method.

In addition to accuracy gains, the value of P was chosen to be 20 because of image reconstruction with a subset of eigenvectors. For example, given the mode shape shown in Figure 5.1, the top 20 eigenvectors are given in Figure 5.5. After 15 modes, it becomes apparent that the mode shape is the same as the one seen in Figure 5.1.

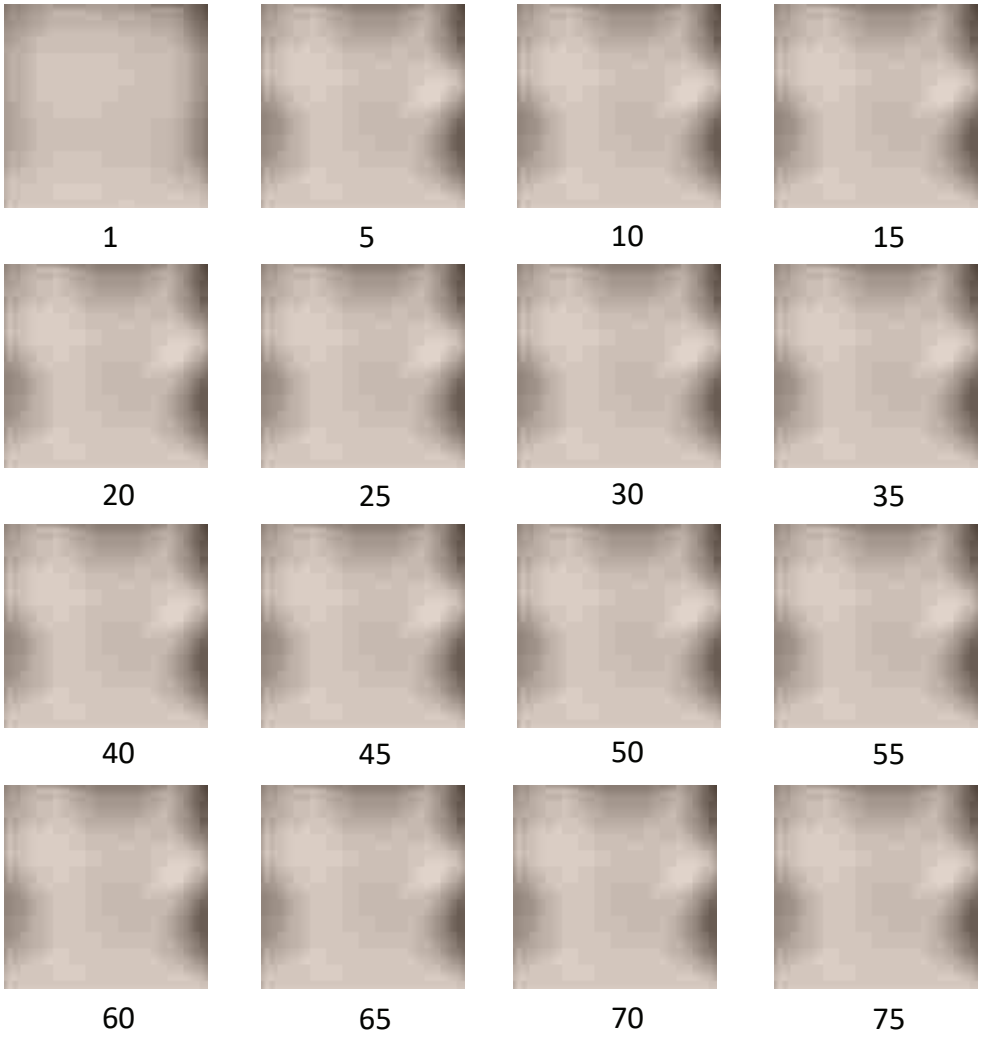

Figure 5.3: PC reconstruction of Figure 5.1. This figure presents 16 of 75 images that were reconstructed from the principal components of a vibrational mode shape. The first few reconstructions do not provide a clear distinction for a human user to identify the mode shape. After about 10-15 mode images, the mode shape starts to become clear.

5.2 Bilinear Eigenblades Results

After SVM was established as the best classifier, a test set of 1,922 images were run through the classifier. This image set only contains images that were identified as a distinct mode shape as described earlier. Each test image was run through the classifier to produce a prediction and probability score. If the probability score was less than 0.50, then the mode shape image was automatically labeled as an "unknown" mode. Conversely, if the probability score is greater than 0.50, then the label given during classification is kept and sorted accordingly.

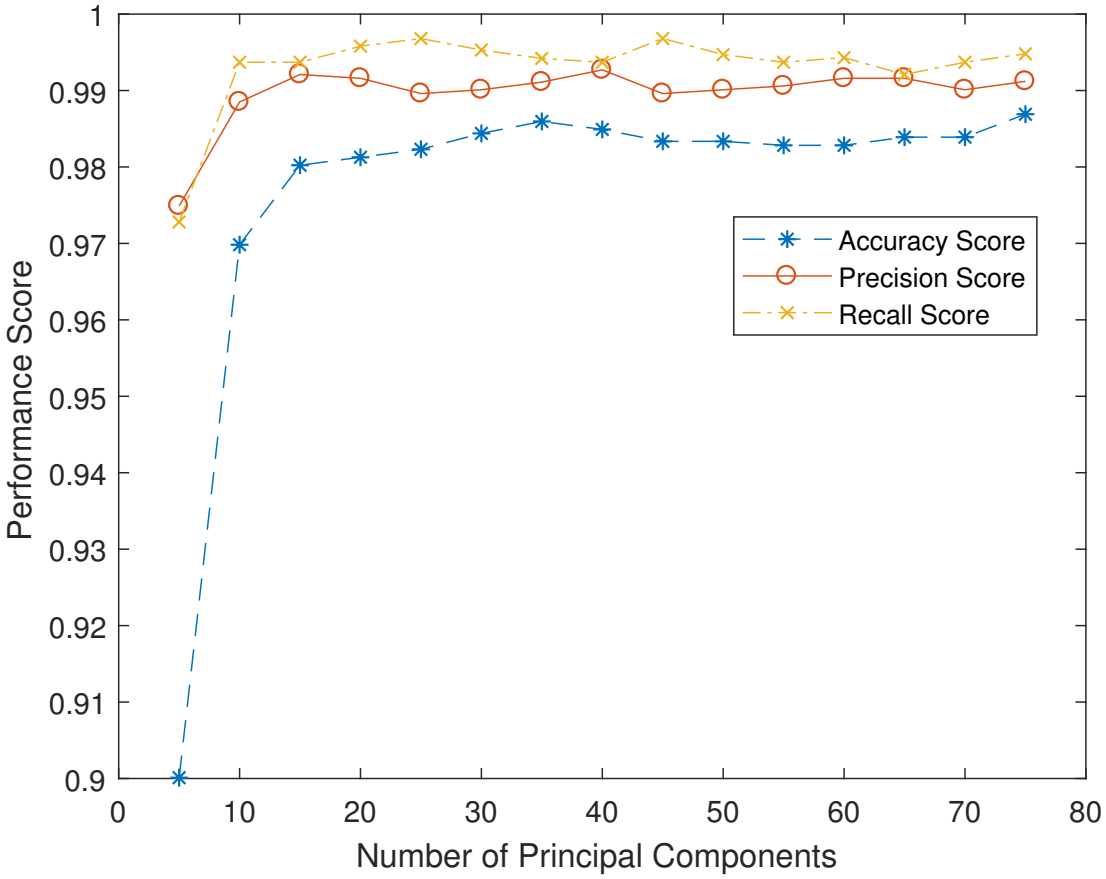

Figure 5.4: Average accuracy, precision, and recall performance scores for the Triangle Eigenblades. After about 20 principal components, there is little gain in accuracy, precision, and recall from the graph. Notice that there is a decreasing rate of return for more PC scores extracted.

The results for this test set are given in Table 5.3. This table shows the accuracy, precision, and recall for each mode. In addition, global accuracy, precision, and recall averages are reported in the same table. A confusion matrix was generated for this multi-class classification problem, which facilitated the performance evaluation of the "Eigenblades" method. The confusion matrix for the test set using the bilinear warp is shown in Appendix A.

The accuracy, precision, and recall scores shown in Table 5.1 demonstrate the excellent results of using PCA and SVM to identify mode shapes. As shown in this table, modes 1, 4, 11, and 13 scored a perfect classification for the test set. Most modes' accuracy measured greater than 90%, except mode 12. Where the scores were perfect, the classifier was able to correctly identify all of their respective mode shapes from the test set. This meant that the training space for these

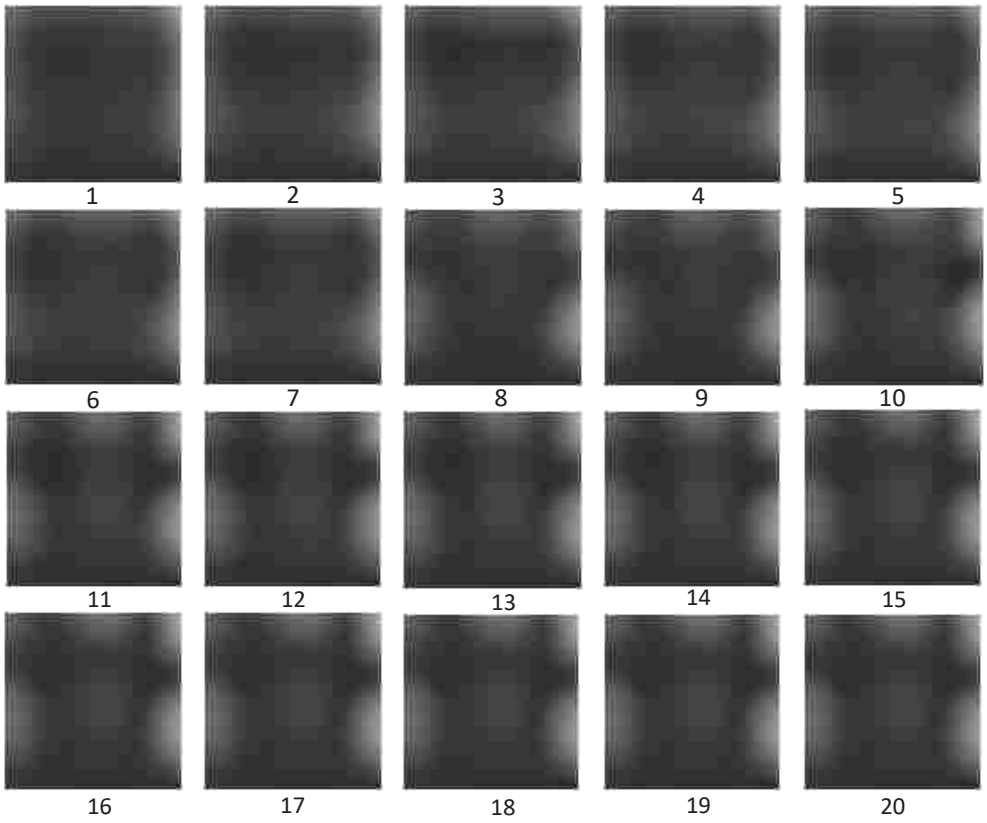

Figure 5.5: Eigenvectors of Figure 5.1. This figure defines the first 20 principal eigenvectors of a vibrational mode shape. The first few eigenvectors do not provide a clear distinction for a human user to identify the mode shape. After about 10-15 mode eigenvectors, the mode shape starts to become clear.

Table 5.1: Accuracy, precision, and recall scores for bilinear Eigenblades classifier. This classifier extracted 70 principal components and was deemed the best of all other cases.

Mode	Accuracy	Precision	Recall
1	1.0000	1.0000	1.0000
2	0.9965	1.0000	0.9965
3	0.9900	0.9900	1.0000
4	1.0000	1.0000	1.0000
5	0.9372	1.0000	0.9372
6	0.9324	0.9415	0.9897
7	0.9949	0.9945	1.0000
8	0.9474	0.9934	0.9558
9	0.9286	1.0000	0.9286
10	0.9876	0.9937	0.9937
11	1.0000	1.0000	1.0000
12	0.8696	0.9756	0.8889
13	1.0000	1.0000	1.0000
Average	0.9844	0.9906	0.9937

modes were complete for this test set. For mode 12, since this was a more complex mode shape, it will need more training data to properly cover the training space.

Another factor that improved the accuracy of this method was removing the need for NURBS interpolation. As discussed in Chapter 2, the NURBS interpolation method showed it was difficult to choose common points for interpolation between two modal displacement vectors as shown in Selin's [10] and Porter's et al. [11] methods. Because the "Eigenblades" method does not use any interpolation of the data, rather dimensionality reduction through PCA, the main components of the data are kept intact. In addition to good accuracy, this method demonstrated that it could correctly classify more modes than prior research. Both [10] and [11] classified only five mode shapes. In this work, 13 were achieved. A larger training set could increase the number of different mode shapes that could be classified.

As discussed previously, the threshold value for the probability score was set at 0.50 after empirical evidence suggested that this value was a good compromise between confident classification and generalization. Table 5.2 shows the average accuracy, precision, and recall results for the test set with different threshold values. In each test, the number of PCs, P, extracted was 20. It can be shown that the threshold value of 0.50 does not give the best results, as can be seen when the threshold is 0.05, the accuracy score is 0.9875. This means that the probability or confidence of the classifier only needs to be 5% sure before labeling a test mode image a particular class. However, since this test data set is a collection of mode shape images that can be identified with a human with high confidence, these results should be expected. To achieve greater generality, however, a threshold value of 0.50 should be used to maximize accuracy, precision, and recall simultaneously. In addition, the value of 0.50 should be used because it balances the confidence of a classifier versus how much to trust a label the classifier reports.

5.2.1 Limitations of Bilinear Warping

Further experiments with this classifier have shown limitations with this work. First, each image is preprocessed prior to dimensionality reduction with PCA. Part of this preprocessing is using the bilinear warp. The limitation in this algorithm is it only works well with straight edges on an image. Figure 5.6 depicts a mode shape before and after a bilinear warp. Notice the left mode shape image has a curved top edge. After the bilinear warp, the right image shows the extra

Table 5.2: Average accuracy, precision, and recall of the test set at different probability threshold values for bilinear Eigenblades. To balance then need for confident classification and generalization, the threshold value of 0.50 was found to be the best for generalization.

Threshold	Accuracy	Precision	Recall
0.05	0.9875	0.9875	1.0000
0.25	0.9869	0.9885	0.9984
0.50	0.9844	0.9906	0.9937
0.75	0.9662	0.9936	0.9723
0.95	0.8059	0.9949	0.8093

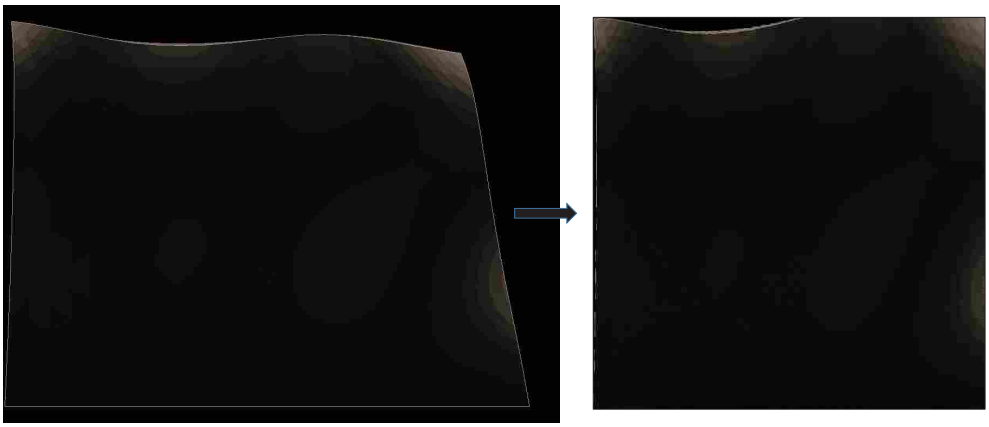

Figure 5.6: Limitations of a bilinear warp for a non-linear edged shape. Notice the extra white space and cutoff at concave and convex locations, respectively after warping of an example mode shape image.

white space and missing details that were cut off. This error occurred because of the curvature of the edges of the mode shape. A potential solution was explored with the triangular warping, as was discussed, to remove the excess white space that caused errors in PCA feature extraction. This would allow blade shapes that were generated from different airfoil shapes, modes with large edge displacements, or blade parameters to be identified using the Eigenblades method with the current training data.

The time to preprocess, PCA feature extract, and then train the SVM classifier was about 385 seconds. Given about 5,300 images within the training set, it took on average .07 seconds per image to create the classifier. Given 1,922 images in the test set, it took an average of 147 seconds to classify them. On average, this equals about .08 seconds per classification.

Table 5.3: Accuracy, precision, and recall for the mode shapes that were in the test set using the triangle warping method. This classifier extracted 20 principal components and was deemed the best of all other cases.

Mode	Accuracy	Precision	Recall
1	1.0000	1.0000	1.0000
2	0.9955	1.0000	0.9965
3	1.0000	1.0000	1.0000
4	1.0000	1.0000	1.0000
5	0.9468	1.0000	0.9468
6	0.9372	0.9417	0.9949
7	0.9949	1.0000	0.9949
8	0.9823	1.0000	0.9823
9	0.9761	1.0000	0.9761
10	0.9876	0.9937	0.9937
11	1.0000	1.0000	1.0000
12	1.0000	1.0000	1.0000
13	1.0000	1.0000	1.0000
Average	0.9863	0.9950	0.9912

5.3 Triangular Eigenblades Results

Like the previous section, which used the bilinear warper, an experiment was performed using the probability threshold for SVM set to 0.50 using the triangle interpolate warper. The results of this experiment are presented in Table 5.3. The confusion matrix is also presented in Appendix A.

The accuracy, precision, and recall scores given in Table 5.3 demonstrate the excellent results of this method compared to the Bilinear Warp as the image preprocessor. This table reports that modes 1, 3, 4, 11, 12, and 13 achieved perfect scores when performing mode shape identification. In addition, all modes but 5 and 6 achieved better than .95 accuracy. The precision and recall values also attest to the superior results of using the triangle warper compared to the bilinear warp. The work is again limited in that conflation still occurs between modes 5 and 6 as seen in Figure 5.11. From this, it can be concluded that the principal modes of vibration from modal analysis might be close enough to be considered the same mode. More work should be done to investigate the natural frequencies, in addition to inspecting the 3D models within NX and ANSYS.

The SVM probability threshold value as mentioned previously was set to 0.50. It was suggested from empirical evidence prior to this work that this value would balance the need for

Table 5.4: Average accuracy, precision, and recall of the test set at different probability threshold values for the Triangle Interpolate Eigenblades method. To balance then need for confident classification and generalization, the threshold value of 0.50 was found to be the best.

Threshold	Accuracy	Precision	Recall
0.05	0.9104	0.9386	0.9717
0.25	0.9154	0.9519	0.9634
0.50	0.9863	0.9950	0.9912
0.75	0.8525	0.9965	0.8558
0.95	0.9096	0.9382	0.9713

confident classification and generalization. A study was performed as part of this research to find the best value for this threshold when the number of PCs extracted was set to 20 and the triangle division value, D, was also set to 20. Table 5.4 gives the average accuracy, precision, and recall scores for five different threshold values. It can be seen that the value of 0.50 gives the best performance measures as hypothesized.

Just like the previous experiment using the bilinear warping method required extracting 20 PCs, the tests using the triangle warping method have shown that a P value of 20 is also sufficient to classify a mode shape image with similar accuracy. Though there is complexity added to the warping algorithm, the number of PCs needed stays the same. In addition, because the bilinear warp only works well for mode shape images with straight edges, it is limited in its application to other test sets. The triangle warping algorithm would be more general and can be more broadly used for mode shape identification.

The time needed to preprocess the image and train the triangular Eigenblades algorithm took 1264 seconds. Given about 5,300 images within the training set, it took on average .24 seconds per image to create the classifier. Given 1,922 images in the test set, it took on average of 613 seconds to classify the set. On average, this equals about .32 seconds per classification.

With the complexity of the triangle interpolate warp, the amount of time needed to process an image increases by about 400%. However, the benefits of triangle warp method to transform complex mode images that had curved edges on a quadrilateral shape outweighs the cost. Figure 5.7 shows the same mode image used to describe the limitations of the bilinear warping method. After using the triangle warping method, the amount of white-space error is reduced significantly.

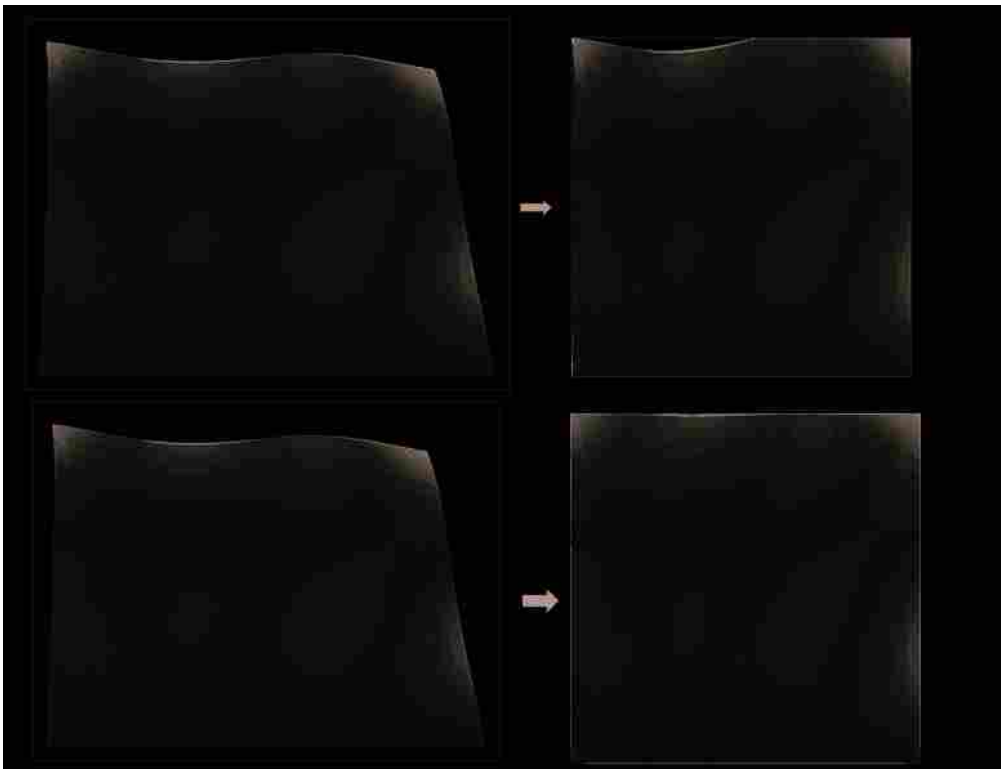

Figure 5.7: Limitations of the bilinear warp method for a non-linear edge shape improved by the triangle interpolate warping. Notice how the tip of the blade has the white space removed and therefore, less error during feature extraction.

In addition to this one example, after investigating some of bilinear Eigenblades and triangle interpolate Eigenblades classification results, it was apparent that the triangle warping had better accuracy performance for the test set. For example, Figure 5.8 shows a set of mode 8's that were incorrectly classified by the bilinear Eigenblades algorithm, but were classified correctly by the triangle Eigenblades warping method. In each of these mode images, there is a large amount of curvature on each of the non-fixed edges. Examples like these show the utility of the triangle warp method in the Eigenblades algorithm.

5.3.1 Limitations of Triangle Interpolate Warping

The triangle warping algorithm described in Chapter 3 and 4 produced comparable results to the bilinear warping algorithm with equals number of PCs to describe an image. The added complexity however, significantly increases the time needed to classify each image. Table 5.5 shows a comparison of computation time for the bilinear warping classifier and the triangle warper

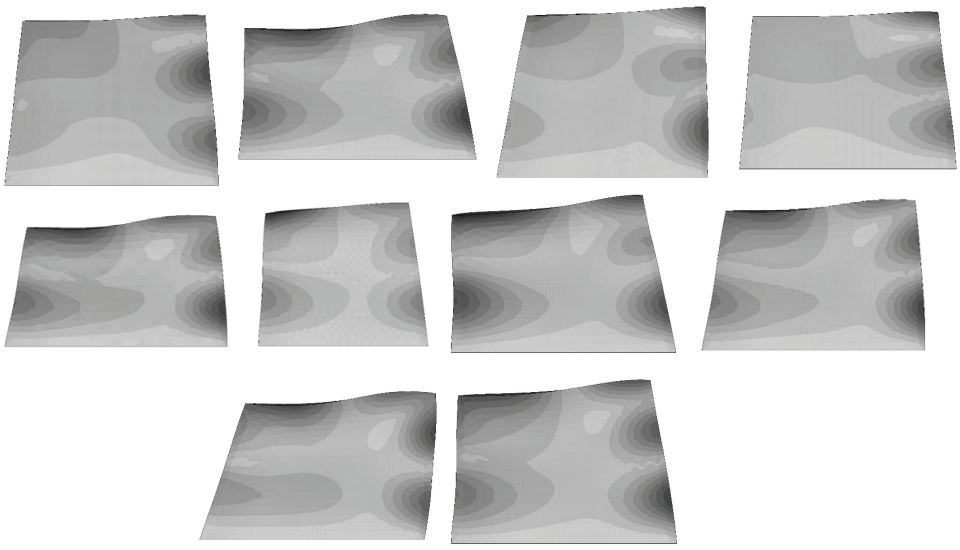

Figure 5.8: Set of mode 8 images where the bilinear warp fails and the triangle warp does well. Notice that each of these images has edges that are non-linear. After feature extraction with PCA, these images were correctly identified using the triangle interpolate warping algorithm, whereas they were incorrectly classified by the bilinear warping Eigenblades.

Table 5.5: Comparison of average computation times per image for bilinear and triangular warping algorithms.

Warping Algorithm	Training (seconds)	Testing (seconds)
Bilinear	0.07	0.08
Triangle Interpolate	0.25	0.32

classifier. Though more accurate, the triangle interpolate warping algorithm takes approximately four times longer than the bilinear warping algorithm to produce relatively similar results in accuracy, precision, and recall. Additionally, comparing the reconstructions shown in Figure 5.1 and Figure 5.5, shows that the mode image in both seem to be clear around 15 modes. This is to be expected as noted by Turk and Pentland [42]. They showed that an infinite number of PCs can perfectly recreate a face image, but there is a value that defines a "good-enough" point, where a face can be easily recognized. For the warping algorithms, even with the more accurate triangle interpolate warp, there is a point that is good enough.

The reason the triangle interpolate warping algorithm was created was to resolve the error/white-space created using the bilinear warping method on mode images with large curvatures on the

edges. For the most part, the triangle interpolate warping performs well as seen in the previous examples. However, with more extreme examples, as seen in Figure 5.9, the triangle warping can fail. This figure depicts extreme curvatures that causes the equally spaced triangles to intersect the white-space and each other. In cases like these, the triangle interpolate warping method would fail to produce human readable and accurate modes after warping.

Possible solutions to this problem is to convert all white space within the image to the mean grayscale value, 127. This would then allows the mean of the image to not be affected by the extra white space that is left over after triangle warping. In addition, another solution could be to divide the mode image into separate quadrants, where each quadrant performs its own triangle interpolate warp. Afterwards, each section would be combined to create the full mode image. However, both these possible solutions have their own limitations and possible errors that could be created. Additional research should be done in this area to expand the triangle warping algorithm to include mode images that have large amounts of curvature along the edges.

5.4 Comparison to Prior Work

The work performed previously to recognize mode shape images by Selin [10] and Porter et al. [11] demonstrated that mode shapes could be distinguished by comparing the nodal displacement of the blades. The three blades geometries used to test their methods are shown in Figure 5.10, which include a rectangular plate, tapered twisted linear plate, and tapered twisted non-linear plate. Of the three, the tapered twisted non-linear plate was the most complex and most difficult to identify.

Therefore, the tapered twisted non-linear plate was used as the dominating example. Research by Selin and Porter et al. demonstrated how blade geometry changed from the nominal during Latin-hypercube sampling. In Selin's case, the maximum change was at maximum 40% from nominal, whereas Porter et al. showed a variation of 100% from nominal. Selin was able to accurately recognize 10 modes shapes, where Porter showed results for 8 modes.

In this research, all mode images were tapered twisted non-linear airfoils and were given a design space of up to 100% from the nominal. In addition, this research had 16 modes that could be identified by the classifier. In general, this research was compared to the "worse-cases" of Selin and Porter et al. Table 5.6 compares the global accuracy (averaged accuracy of all mode shapes) of

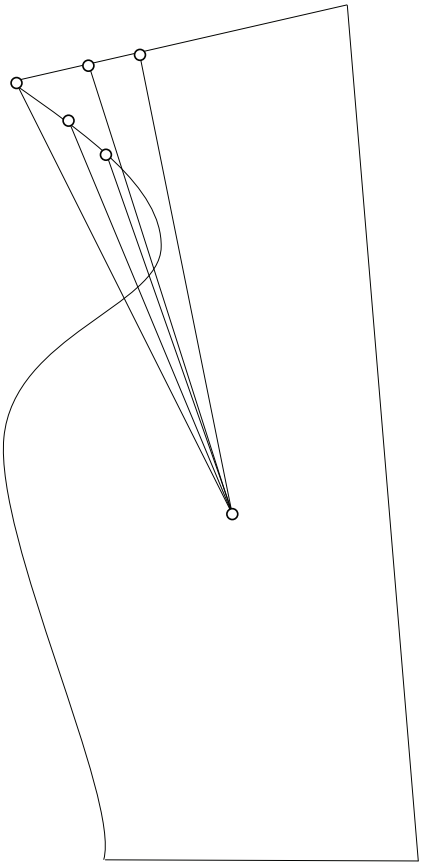

Figure 5.9: Limitations of the triangle interpolate warping method. When there are extreme curvatures on the edges of the mode image, there will be errors created from overlapping triangles, and limitations in interpolation.

Table 5.6: Comparison of results to prior work. This work showed significant improvement over the mode shape identification methods presented by Selin and Porter et al.

Method	Accuracy	Time to recognize a sample (seconds)
Selin	82% (at 40% variation)	1.55
Porter et al.	84% (at 100% variation)	0.32
Bilinear Eigenblades	98.4% (at 100% variation)	.15
Triangular Eigenblades	98.6% (at 100% variation)	.56

the three studies considered. The computational time of these are also compared in Table 5.6. The time for this research and Porter et al. includes the time needed to train the classifier and classify an unknown mode image.

Both this research and Porter's shows significant improvement in both accuracy and speed compared to Selin's research. The bilinear and triangular warping algorithm performed better

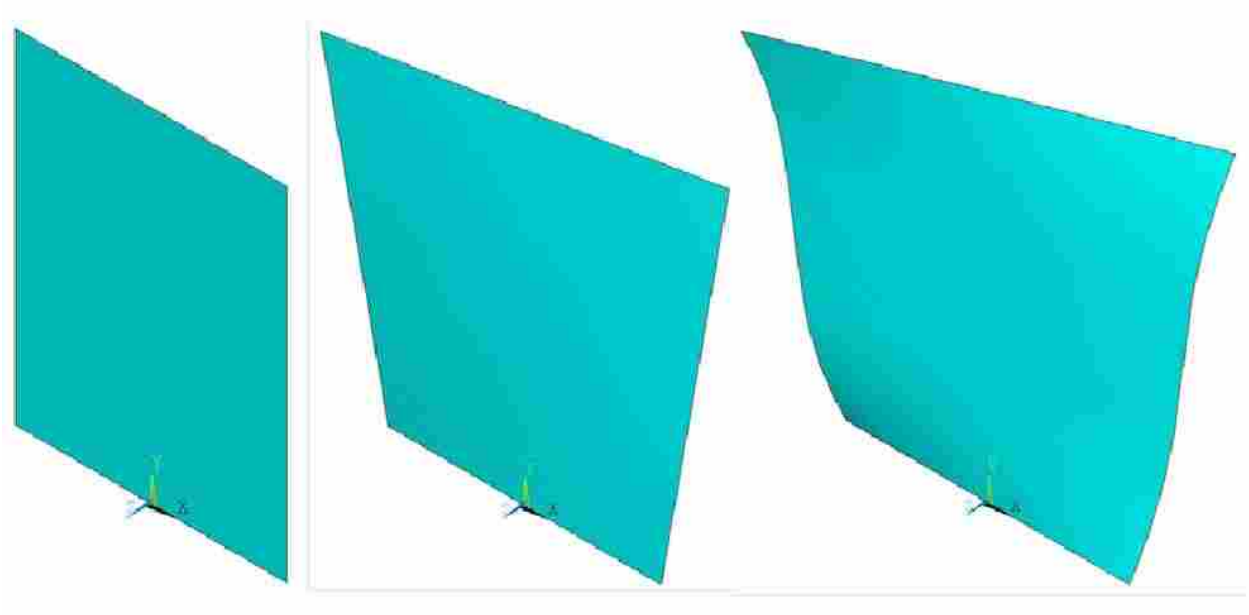

Figure 5.10: The geometry used by both Selin's and Porter et al,'s work. This geometry was used to generate modal displacement vectors to recognize mode shapes from displacements.

than Porter's work, however, only the bilinear warp performed faster. Selin was able to classify accurately up to 10 different mode shapes, whereas Porter et al. was limited to 8 mode shapes. In this research, up to 16 modes could be classified, and 13 were accurately classified from the test set. This demonstrates that using the Eigenblades method is a viable replacement and dominant solution over the others in terms of accuracy and computational speed improvements.

5.5 Overall Scope and Limitations

There were 13 unique modes that were found in the test set, though 10 modes were extracted for each blade. This suggests that the problem of open-set recognition [76] might be an issue, if there are not enough mode shapes trained. Again, more mode shapes need to be trained in order to avoid this problem. The number of modes required should be determined by the user and their analysis needs.

For both the bilinear and triangle warping methods, similar performance was achieved while classifying the test set. However, the time needed to train the classifier and to classify the unknown mode images were very different. With the triangle warping algorithm, it took an average about four times longer to train and classify the mode shape images for accuracy gains, which

were less than .1%. However, given that these mode images could be more complex, the triangle interpolate Eigenblades method is still the better candidate to perform mode shape recognition. The reason this work considered using the image warper was so mode images that were dissimilar to the training set could be identified with the trained classifier. More work should be done to create a test set that is significantly dissimilar to the training set to perform a true test on the triangle warper.

The number of the training images this work was based on depended on how many DOEs were run prior to testing the classification algorithm. Creating more training images would be as simple as running another DOE and having a subject matter expert choose more training images for each distinct mode. This would be a simple next step to increase the accuracy. In addition, other mode shapes or blade geometries could be added to this training set for a more diverse training repository.

Further inspection of the confusion matrix for modes 5 and 6 showed one reason for the inaccuracy was ten images that were supposed to be classified as mode 6 were misclassified as mode 5. Looking at the images directly, it was noticed that the errors were due to image conflation, as seen in Figure 5.11. All mode shapes in this figure are either mode 5 or 6, where mode 5 is labeled as $\boldsymbol{a}$ and mode 6 is labeled as $\boldsymbol{h}$. The images labeled $\boldsymbol{a}$ and $\boldsymbol{h}$ are very distinct mode shapes. However, the distinction begins to weaken as the image chain progresses. Image $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{d}$, still look a lot like mode 5, but the clarity of the mode begins to weaken. The same could be said about image $\boldsymbol{g}$ and $\boldsymbol{f}$ compared to image $\boldsymbol{h}$. Image $\boldsymbol{e}$ looks similar to both image $\boldsymbol{d}$ and $\boldsymbol{f}$. However, because of this lack of distinction, it is difficult even for a human expert to determine which mode shape image $\boldsymbol{e}$ should be given.

Both warping algorithms assumes that a mode shape image is a general quadrilateral shape. This is a fair assumption to make for these general compressor blades in a jet engine, but both warping algorithms are limited to shapes that have four corners and edges. Other shapes that are not quadrilateral, or have distinct corners and edges, were not considered in this research.

5.6 Varying Image Down-sampling Resolution

Throughout this work, the down-sampled mode images were limited to a 100×100 pixel squares. The value of 100 was chosen to reduce the image down to a manageable size for pre-

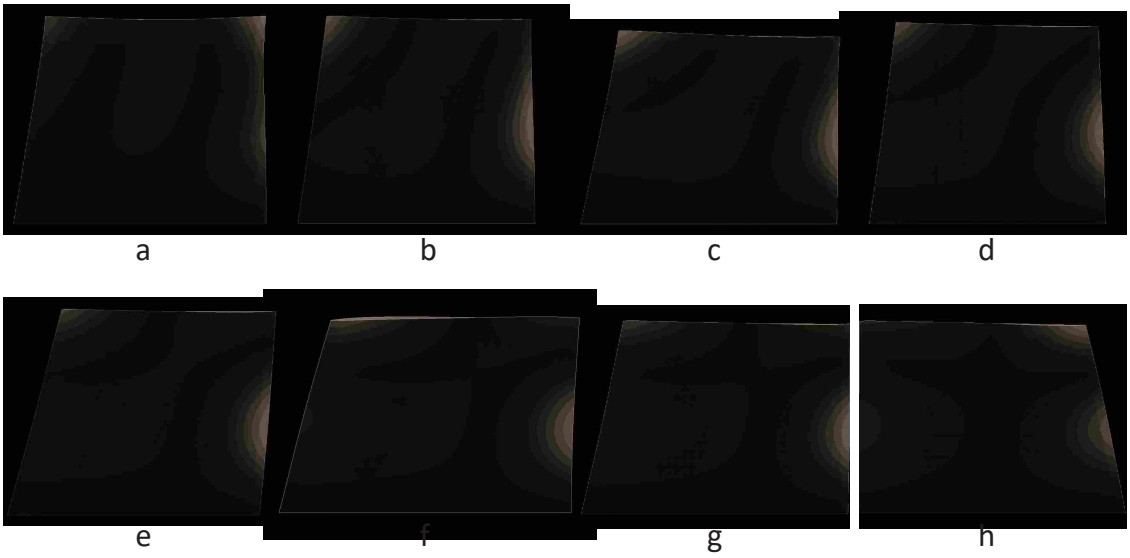

Figure 5.11: Conflation of mode 5 to mode 6. Mode 5 is seen in image ***a*** and mode 6 is image ***h***. These two images are distinct mode shapes. However, conflation can be seen in the images as different geometries are considered. These images are difficult even for human experts to classify and that is why they were incorrectly labeled.

processing while keeping it large enough for a human to classify an image. In this section, this research explores how the average accuracy, precision and, recall change when the image size differs from 100×100. In addition, the average computational time is also reported. Given these values, a more intelligent decision can be made on the down-sampling size of a mode image. It should be noted that these performance values should be compared in relation to each pixel size and Eigenblades method because the performance on the computer used to perform these experiments will vary from others. The computational resources that performed these experiments was assumed to stay constant throughout all the experiments.

5.6.1 Bilinear Eigenblades

First, the bilinear Eigenblades method was tested to find the optimum down-sampling size. Table 5.7 shows the down-sample size, average performance measure scores, and the computational time needed to identify the test set. For computation time, the average is calculated per image of time needed to both train and identify a test image. It should be noted that these values are averaged over the entire training and test set and are not mode/class specific. In this table, the

Table 5.7: Average performance of Bilinear Eigenblades at different down-sampling pixel values. The best accuracy comes from the 100×100 pixel image. The two largest image sizes failed because of computer memory limitations.

Pixels	Accuracy	Precision	Recall	Time per image (train)	Time per image (test)
10×10	0.9751	0.9935	0.9812	0.041	0.052
50×50	0.9818	0.9937	0.9880	0.045	0.052
100×100	0.9844	0.9906	0.9907	0.070	0.080
200×200	0.9813	0.9926	0.9885	0.070	0.086
500×500	FAIL	FAIL	FAIL	FAIL	FAIL
1000×1000	FAIL	FAIL	FAIL	FAIL	FAIL

value of P was set to 20 and the probability threshold was held at 0.50. This allows for a direct comparison of performance to the 100×100 case.

First, the 500×500 and 1000×1000 sub-sampled image experiments were unable to finish because the workstation that the classification was performed on ran out of memory each time an attempt was made to create a classifier. It can then be concluded that these sizes and larger are computationally expensive and should be avoided. It comes to show from the rest of the data that the other sizes still perform rather well, both in accuracy, precision, recall, and computational time. For the performance measures, that maximum difference between each of the scores is about .1%. The time needed to train and classify the Eigenblades classifier were significantly less for the 10×10 and 50×50 pixel images. Overall, the Eigenblades algorithm still performed the best in terms of classification accuracy using the 100×100 sized images while using the bilinear warping method.

5.6.2 Triangle Interpolate Eigenblades

Similar to the bilinear Eigenblades method, Table 5.8 shows the performance of the triangle interpolate Eigenblades method at different sampling pixel values. Again, the number of PCs, P, was held at 20 and the probability threshold was set to 0.50 to allow for a direct comparison to the previous experiments.

Like the Bilinear Eigenblades, this method also fails when the image size is set to 1000×1000. Again, this was due to memory failure. The reason why the 500×500 image size experiment succeeded was because it was performed after a fresh restart of the machine when nothing else was

Table 5.8: Average performance of Triangle Eigenblades at different down-sampling pixel values. The best accuracy comes from using the largest images. It would seem that the Triangle Interpolate Eigenblades does not experience the over-fitting issue.

Pixels	Accuracy	Precision	Recall	Time per image (train)	Time per image (test)
10×10	0.9886	0.9927	0.9958	0.209	0.249
50×50	0.9891	0.9932	0.9958	0.193	0.230
100×100	0.9863	0.9950	0.9912	0.240	0.320
200×200	0.9896	0.9927	0.9966	0.300	0.284
500×500	0.9922	0.9922	0.9989	5.938	0.706
1000×1000	FAIL	FAIL	FAIL	FAIL	FAIL

running. However, it did give the best accuracy results of all of the experiments. It would seem that the larger images are able to produce PC scores that enables the classifier to make better decisions. However, with the small accuracy gain, the amount of time to train a classifier increases by an order of magnitude and the time to classify an image more than doubles the next largest value. As expected, a trade-off is expected with larger images. The user must either choose better classifier performance or computational performance. Also, unlike the Bilinear method, the accuracy continues to increase. This could mean that the larger images are better able to create a feature vector that defines the feature space of a particular mode shape.

5.7 Varying the Number of Training Images

The general assumption that more training images equates to better performance was tested as part of this research. Given that the lead time to create a Eigenblades classifier can be a constraint, how the number of training images for each mode shape is important to know to reduce the overhead costs of creating a mode shape identifier. This section shows how the performance changes with varying number of PCs required for different number of training mode images. For both Eigenblades methods, mode 11 is removed from training and testing because it does not contain a significant number of training and test images. The figures and the tables report the average performance of the classifier for a given percentage of training images that were reported in Chapter 4. For each Eigenblades classifier, each image was reduced to a 100×100 sized image. In addition, the probability score threshold was set to 0.5. Thus, allowing for a direction comparison to the experiments that were performed previously with 100% of the training data.

Table 5.9: Average performance of the Bilinear Eigenblades method at varying training image quantities at best PC score, $P = 20$. The more training data there is, the better the accuracy, but with increased computation time.

Percentage	Accuracy	Time per image (train, sec.)	Time per image (test, sec.)
10	0.8731	0.0037	0.0532
25	0.9251	0.0098	0.0528
50	0.9547	0.0221	0.0751
75	0.9781	0.0350	0.0571
90	0.9849	0.0435	0.0601

5.7.1 Bilinear Eigenblades

The Bilinear Eigenblades method was first used to test the reduced training sets mode images. Figure 5.12, shows how the performance varies with the different number of PCs extracted. Each line in this figure represents a different percent of training data from the original full training set. As shown in Figure 5.12, in general, the larger percentage of training data used, the more accurate the classifier. The line outlined by the diamond shape uses 100% of the training data. Every other line never surpasses the 100% training line. Therefore, it is important to have a large training set to maximize the accuracy. It is interesting to see that all lines except for the 10% line decreases in accuracy around 15-20 PCs. As explained previously, after 20 PCs, the classifier could be over-fitting the training data and perhaps even fitting the white-space error, thus causing the decrease in accuracy.

In addition, the performance measures are presented in Table 5.9. This table presents the performance measures and the time required for each of training image percentage at the best PC value, $P = 20$.

With the simplicity of the bilinear warp, it can be seen that most of the computational expense comes from training the SVM classifier. From this table, it seems the training time is proportional to the number of training images. It will then be necessary for the user of the Eigenblades method to determine whether computational speed or classification accuracy is more important in their application.

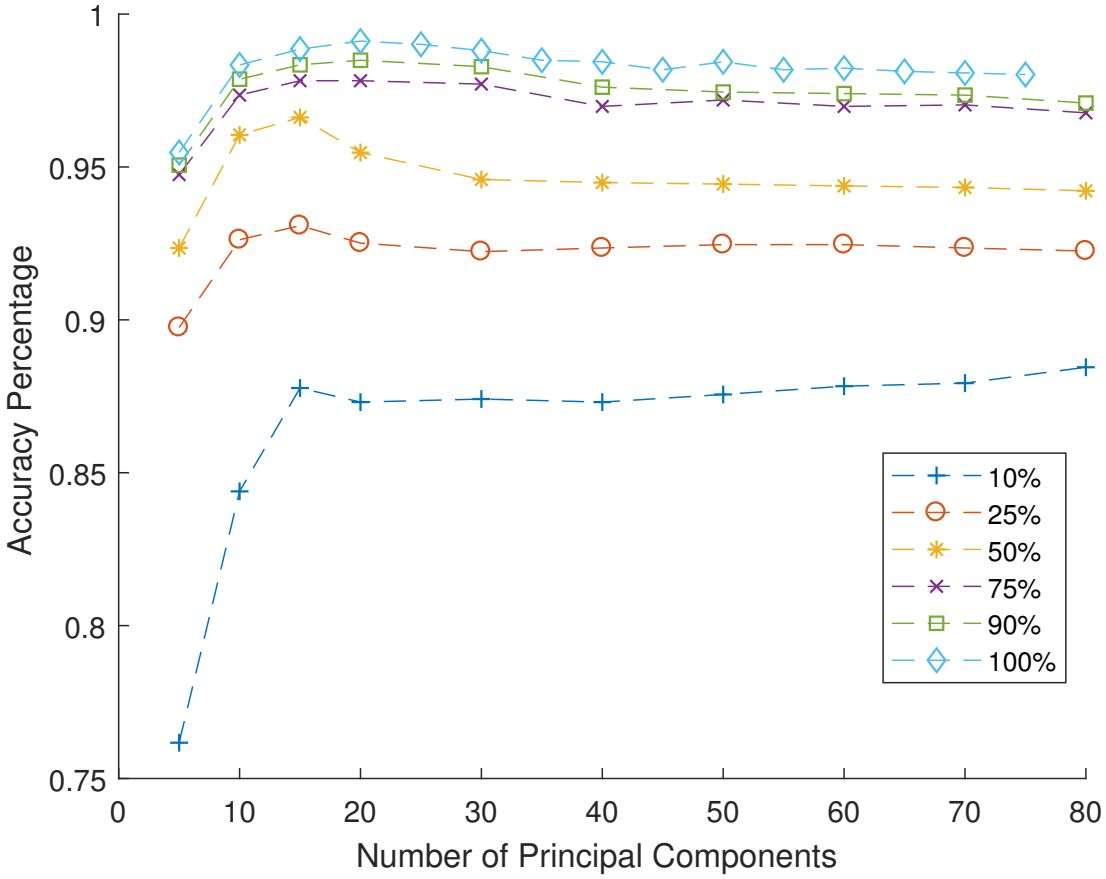

Figure 5.12: Performance of the Bilinear Eigenblades classifier at varying training image percentages. The more training images there are, the better the classifier. However, after about 20 PCs, the accuracy measure starts to decrease. This could be over-fitting of the training data.

5.7.2 Triangle Interpolate Eigenblades

Similar to the Bilinear Eigenblades, the Triangle Interpolate Eigenblade's performance is shown in Figure 5.13, where each line represents a different percentage of training images from the original. As explain previously, the number of training images significantly affects the quality of the Eigenblades classifier. However, unlike the Bilinear Eigenblades, the 75%, 90% and 100% classifiers are very close to each other when it comes to classification accuracy. After about 20 PCs, the number of training data needed to gain high accuracy converges for these three lines. This could mean that the Triangle Interpolate Eigenblades is better able to recognize complex features using less training data. In addition, the accuracy scores reach a steady state, and do not tend to

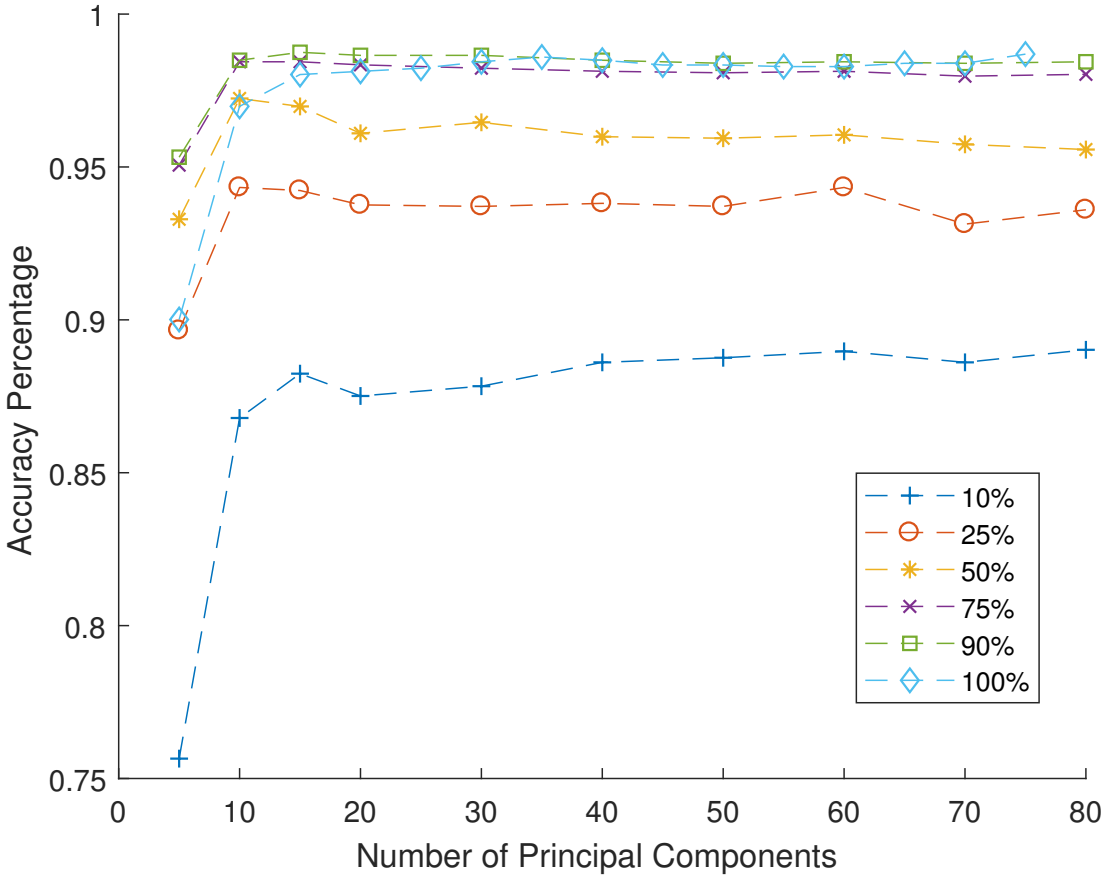

Figure 5.13: Performance of the Triangle Interpolate Eigenblades classifier at varying training image percentages. After about 20 PCs, the 75%, 90%, and 100% lines are about equal.

decrease, again implying that the features vectors created by the Triangle Interpolate Eigenblades do not try to fit too much white-space error.

In addition, the performance measures are presented in Table 5.10. This table presents the performance measures and the time required for each training image percentage at the best PC value. As the number of training images increases, so does the time needed to train the classifier. It is interesting to note that the time needed to classify an unknown mode image is independent of the number of training images, as expected.

Table 5.10: Average performance of the Triangle Interpolate Eigenblades method at varying training image quantities at best PC score.

Percentage	Accuracy	Time per image (train, sec.)	Time per image (test, sec.)
10	0.8751	0.0178	0.2302
25	0.9376	0.0471	0.2285
50	0.9610	0.0950	0.2286
75	0.9834	0.1491	0.2315
90	0.9865	0.1783	0.2302

5.8 Future Work

The methods shown in this work have utilized the classic approach to image recognition and computer vision, that is, preprocessing the image, performing feature vectors for images and then creating machine learning classifier to create a hyper-planes that separates the training data. The results generated in this research have shown that mode shape of turbomachinery blades can be identified using computer vision and machine learning. However, the mode shapes are only limited to those generated from similar DOEs. An extension to this work would include classifying mode images created from other DOEs with different airfoil shapes, non-quadrilateral shapes, and images created from experimental data.

In addition, this work is limited in that only trained images can be recognized. Research into the open set recognition would improve this work by creating new classifiers when the current classifier does not recognize a given test mode.

CHAPTER 6. CONCLUSION

Modal analysis of turbomachinery blades is an integral part of design exploration for jet-engine design. This research has created a method that utilizes image processing, computer vision and machine learning to automatically identify mode shape images created within a design of experiments.

Revisiting the research questions presented in Chapter 1 shows that this research has answered them as follows:

- Can mode shape identification created within a DOE be performed using images?
  - Yes. The Eigenblades classification method was created during this research that accurately identified mode shape images that were created from a DOE. In addition to accuracy, the computational cost required for Eigenblades performed better than prior work.
- If so, how will it work?
  - Using a standard computer vision approach, a machine learning classifier was created using a training set of mode shape images. The Eigenblades classifier was able to accurately and quickly classify mode images of an unknown mode shape set.
- Given that is works, how does it compare to prior work in mode shape identification in accuracy and computation performance?
  - This research showed that mode shape identification using images performs better than previous work, in both accuracy and time needed to classify an image.
- How does image warping affect the results of mode shape identification? Does it help?

– In general, warping the image does improve the accuracy of identifying mode images, however, there is an associated computational cost. Two warping methods were investigated. The bilinear warp shows exceptional performance in accuracy and computation speed. The triangular interpolate warping method was created in this research to facilitate warping of non-linear edges on blades.

REFERENCES

[1] Jansen, B., 2016. Ntsb: Southwest engine failure lost fan blade, Sep. vii, 2

[2] Flack, R. D., 2005. *Fundamentals of jet propulsion with applications*., Vol. 17 Cambridge University Press. 1

[3] Cowles, B., 1996. "High cycle fatigue in aircraft gas turbines: an industry perspective." *International Journal of Fracture,* **80**(2-3), pp. 147–163. 1

[4] Saravanamuttoo, H. I. H., Rogers, G. F. C., and Cohen, H., 2001. *Gas turbine theory*. Pearson Education. 1

[5] Maktouf, W., and Sai, K., 2015. "An investigation of premature fatigue failures of gas turbine blade." *Engineering Failure Analysis,* **47**, pp. 89–101. 1

[6] Williams, T., 2016. Investigative update provides initial findings in investigation of uncontained engine failure, Sep. 1

[7] Chopra, A. K., 2001. *Dynamics of structures: theory and applications to earthquake engineering*. Prentice-Hall. 1

[8] Poursaeidi, E., Babaei, A., Arhani, M. M., and Arablu, M., 2012. "Effects of natural frequencies on the failure of r1 compressor blades." *Engineering Failure Analysis,* **25**, pp. 304–315. 1

[9] Ewins, D. J., 1984. *Modal testing: theory and practice*., Vol. 15 Research studies press Letchworth. 3, 7

[10] Selin, E. D., 2012. "Application of parametric nurbs geometry to mode shape identification and the modal assurance criterion.". 4, 5, 8, 23, 69, 75

[11] Porter, R., Hepworth, A., and Jensen, C. G., 2016. "Application of machine learning and parametric nurbs geometry to mode shape identification." *Computer-Aided Design and Applications,* **13**(2), pp. 184–198. 4, 5, 8, 9, 23, 69, 75

[12] Guillaume, P., 2007. "Modal analysis." *Department of Mechanical Engineering, Vrije Universiteit Brussel, Pleinlaan,* **2**, pp. 4–6. 5

[13] Duffy, K. P., Bagley, R. L., Mehmed, O., and Choi, B., 2000. "On a self-tuning impact vibration damper for rotating turbomachinery.". 5

[14] Gonzalez, R., and Wintz, P., 1977. "Digital image processing.". 5, 29

[15] Allemang, R. J., 2003. "The modal assurance criterion–twenty years of use and abuse." *Sound and vibration,* **37**(8), pp. 14–23. 7, 8

[16] Palm, W. J., 2005. *System dynamics*. McGraw-Hill Higher Education. 7

[17] Allemang, R. J., and Brown, D. L., 1982. "A correlation coefficient for modal vector analysis." In *Proceedings of the 1st international modal analysis conference*, Vol. 1, Orlando: Union College Press, pp. 110–116. 8

[18] Piegl, L., and Tiller, W., 2012. *The NURBS book*. Springer Science & Business Media. 8

[19] Chaoping, Z., and Yinchao, L., 2012. "Mode shape description and model validation of axisymetric structure [j]." *Journal of Nanjing University of Aeronautics & Astronautics,* **5**, p. 022. 8

[20] Khotanzad, A., and Hong, Y. H., 1990. "Invariant image recognition by zernike moments." *IEEE Transactions on pattern analysis and machine intelligence,* **12**(5), pp. 489–497. 8, 18

[21] Wang, W., Mottershead, J. E., and Mares, C., 2009. "Shape descriptors for mode-shape recognition and model updating." In *Journal of Physics: Conference Series*, Vol. 181, IOP Publishing, p. 012004. 9

[22] Marsland, S., 2015. *Machine learning: an algorithmic perspective*. CRC press. 9, 10, 11, 12, 13, 14, 16, 17, 38, 40, 41

[23] Bishop, C. M., 2006. "Pattern recognition." *Machine Learning,* **128**. 9

[24] Fisher, R. A., 1936. "The use of multiple measurements in taxonomic problems." *Annals of human genetics,* **7**(2), pp. 179–188. 10

[25] Wu, T.-F., Lin, C.-J., and Weng, R. C., 2004. "Probability estimates for multi-class classification by pairwise coupling." *Journal of Machine Learning Research,* **5**(Aug), pp. 975–1005. 10

[26] Hornik, K., Stinchcombe, M., and White, H., 1989. "Multilayer feedforward networks are universal approximators." *Neural networks,* **2**(5), pp. 359–366. 11

[27] Rosenblatt, F., 1958. "The perceptron: A probabilistic model for information storage and organization in the brain.." *Psychological review,* **65**(6), p. 386. 11

[28] Ruck, D. W., Rogers, S. K., Kabrisky, M., Oxley, M. E., and Suter, B. W., 1990. "The multilayer perceptron as an approximation to a bayes optimal discriminant function." *IEEE Transactions on Neural Networks,* **1**(4), pp. 296–298. 12

[29] Read, J., Pfahringer, B., Holmes, G., and Frank, E., 2011. "Classifier chains for multi-label classification." *Machine learning,* **85**(3), pp. 333–359. 14

[30] Gardner, M. W., and Dorling, S., 1998. "Artificial neural networks (the multilayer perceptron)a review of applications in the atmospheric sciences." *Atmospheric environment,* **32**(14), pp. 2627–2636. 14

[31] Beyer, K., Goldstein, J., Ramakrishnan, R., and Shaft, U., 1999. "When is nearest neighbor meaningful?." In *International conference on database theory*, Springer, pp. 217–235. 14

[32] Dudani, S. A., 1976. "The distance-weighted k-nearest-neighbor rule." *IEEE Transactions on Systems, Man, and Cybernetics*(4), pp. 325–327. 15

[33] Garcia, V., Debreuve, E., and Barlaud, M., 2008. "Fast k nearest neighbor search using gpu." In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, IEEE, pp. 1–6. 15

[34] Loncaric, S., 1998. "A survey of shape analysis techniques." *Pattern recognition,* **31**(8), pp. 983–1001. 18

[35] Costa, L. d. F. D., and Cesar Jr, R. M., 2000. *Shape analysis and classification: theory and practice*. CRC Press, Inc. 18

[36] Wang, W., Mottershead, J. E., and Mares, C., 2009. "Vibration mode shape recognition using the zernike moment descriptor." In *Conference proceedings of the society for experimental mechanics series*. 18

[37] Wang, W., Mottershead, J. E., Siebert, T., and Pipino, A., 2012. "Frequency response functions of shape features from full-field vibration measurements using digital image correlation." *Mechanical systems and signal processing,* **28**, pp. 333–347. 18

[38] Wang, W., Mottershead, J. E., and Mares, C., 2009. "Mode-shape recognition and finite element model updating using the zernike moment descriptor." *Mechanical systems and signal processing,* **23**(7), pp. 2088–2112. 18

[39] Wang, W., Mottershead, J. E., Ihle, A., Siebert, T., and Schubach, H. R., 2011. "Finite element model updating from full-field vibration measurement using digital image correlation." *Journal of Sound and Vibration,* **330**(8), pp. 1599–1620. 18

[40] Persoon, E., and Fu, K.-S., 1977. "Shape discrimination using fourier descriptors." *IEEE Transactions on systems, man, and cybernetics,* **7**(3), pp. 170–179. 18

[41] Chen, G., and Bui, T. D., 1999. "Invariant fourier-wavelet descriptor for pattern recognition." *Pattern recognition,* **32**(7), pp. 1083–1088. 18

[42] Turk, M. A., and Pentland, A. P., 1991. "Face recognition using eigenfaces." In *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on*, IEEE, pp. 586–591. 18, 20, 21, 74

[43] Belhumeur, P. N., Hespanha, J. P., and Kriegman, D. J., 1997. "Eigenfaces vs. fisherfaces: Recognition using class specific linear projection." *IEEE Transactions on pattern analysis and machine intelligence,* **19**(7), pp. 711–720. 19

[44] Zuo, H., Lang, H., Blasch, E., and Ling, H., 2017. "Covert photo classification by deep convolutional neural networks." *Machine Vision and Applications,* **28**(5-6), pp. 623–634. 19

[45] LeCun, Y., Bengio, Y., and Hinton, G., 2015. "Deep learning." *Nature,* **521**(7553), pp. 436–444. 19

[46] Al-Waisy, A. S., Qahwaji, R., Ipson, S., and Al-Fahdawi, S., 2017. "A multimodal deep learning framework using local feature representations for face recognition." *Machine Vision and Applications*, pp. 1–20. 19

[47] Možina, M., Tomaževič, D., Pernuš, F., and Likar, B., 2013. "Automated visual inspection of imprint quality of pharmaceutical tablets." *Machine vision and applications,* **24**(1), pp. 63–73. 19

[48] Larese, M. G., and Granitto, P. M., 2016. "Finding local leaf vein patterns for legume characterization and classification." *Machine Vision and Applications,* **27**(5), pp. 709–720. 19

[49] San Biagio, M., Beltrán-González, C., Giunta, S., Del Bue, A., and Murino, V., 2017. "Automatic inspection of aeronautic components." *Machine Vision and Applications*, pp. 1–15. 19

[50] Usman, K., and Rajpoot, K., 2017. "Brain tumor classification from multi-modality mri using wavelets and machine learning." *Pattern Analysis and Applications*, pp. 1–11. 19

[51] Giraldo-Zuluaga, J.-H., Diez, G., Gomez, A., Martinez, T., Vasquez, M. P., Bonilla, J. F. V., and Salazar, A., 2016. "Automatic identification of scenedesmus polymorphic microalgae from microscopic images." *arXiv preprint arXiv:1612.07379*. 20

[52] Otsu, N., 1979. "A threshold selection method from gray-level histograms." *IEEE transactions on systems, man, and cybernetics,* **9**(1), pp. 62–66. 20

[53] Sobel, I., 1990. "An isotropic 3× 3 image gradient operator." *Machine vision for three-dimensional scenes*, pp. 376–379. 20

[54] Bracewell, R. N., and Bracewell, R. N., 1986. *The Fourier transform and its applications*., Vol. 31999 McGraw-Hill New York. 20

[55] Khan, M. A., Khan, T. M., Soomro, T. A., Mir, N., and Gao, J., 2017. "Boosting sensitivity of a retinal vessel segmentation algorithm." *Pattern Analysis and Applications*, pp. 1–17. 20

[56] Jackway, P., 2000. "Improved morphological top-hat." *Electronics Letters,* **36**(14), pp. 1194–1195. 20

[57] Haijun, L., Lingmin, L., and Xianyi, L., 2010. "A novel preprocessing approach for digital meter reading based on computer vision." In *Proceedings of the third international symposium on computer science and computational technology*, Vol. 1, pp. 308–311. 20

[58] Jolliffe, I., 2002. *Principal component analysis*. Wiley Online Library. 21

[59] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E., 2011. "Scikit-learn: Machine learning in Python." *Journal of Machine Learning Research,* **12**, pp. 2825–2830. 22, 61

[60] Antony, J., 2014. *Design of experiments for engineers and scientists*. Elsevier. 23

[61] Stein, M., 1987. "Large sample properties of simulations using latin hypercube sampling." *Technometrics,* **29**(2), pp. 143–151. 26

[62] Cullum, J. K., and Willoughby, R. A., 2002. *Lanczos algorithms for large symmetric eigenvalue computations: Vol. I: Theory*. SIAM. 29

[63] Shi, J., et al., 1994. "Good features to track." In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, IEEE, pp. 593–600. 29, 33, 53

[64] Wolberg, G., 1990. *Digital image warping*., Vol. 10662 IEEE computer society press Los Alamitos, CA. 29

[65] Bradski, G.", 2000.." *Dr. Dobb's Journal of Software Tools*. 30, 53, 56

[66] Jain, A. K., 1989. *Fundamentals of digital image processing*. Prentice-Hall, Inc. 30, 35

[67] Soille, P., 2013. *Morphological image analysis: principles and applications*. Springer Science & Business Media. 30

[68] Canny, J., 1986. "A computational approach to edge detection." *IEEE Transactions on pattern analysis and machine intelligence*(6), pp. 679–698. 32

[69] Suzuki, S., et al., 1985. "Topological structural analysis of digitized binary images by border following." *Computer vision, graphics, and image processing,* **30**(1), pp. 32–46. 32

[70] Samet, H., 1990. *The design and analysis of spatial data structures*., Vol. 199 Addison-Wesley Reading, MA. 34

[71] Domingos, P., 2012. "A few useful things to know about machine learning." *Communications of the ACM,* **55**(10), pp. 78–87. 40

[72] Davis, J., and Goadrich, M., 2006. "The relationship between precision-recall and roc curves." In *Proceedings of the 23rd international conference on Machine learning*, ACM, pp. 233–240. 42

[73] McKay, M. D., Beckman, R. J., and Conover, W. J., 1979. "Comparison of three methods for selecting values of input variables in the analysis of output from a computer code." *Technometrics,* **21**(2), pp. 239–245. 44

[74] Sobel, I., and Feldman, G., 1968. "A 3x3 isotropic gradient operator for image processing." *a talk at the Stanford Artificial Project in*, pp. 271–272. 56

[75] Halko, N., Martinsson, P.-G., Shkolnisky, Y., and Tygert, M., 2011. "An algorithm for the principal component analysis of large data sets." *SIAM Journal on Scientific computing,* **33**(5), pp. 2580–2594. 60

[76] Bendale, A., and Boult, T. E., 2016. "Towards open set deep networks." In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 77

APPENDIX A. CONFUSION MATRIX

	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6	Mode 7	Mode 8	Mode 9	Mode 10	Mode 11	Mode 12	Mode 13	Mode 14	Mode 15	Mode 16	Unknown
Mode 1	278	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 2	0	287	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 3	0	0	200	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Mode 4	0	0	0	191	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 5	0	0	0	0	194	0	0	0	0	0	0	0	0	0	0	0	0
Mode 6	0	0	0	0	10	193	0	0	0	0	0	0	0	0	0	0	0
Mode 7	0	0	0	0	0	0	195	0	0	0	0	0	0	0	0	0	0
Mode 8	0	0	0	0	0	0	0	104	0	0	0	0	0	0	0	0	0
Mode 9	0	0	0	0	0	0	0	0	37	0	0	0	0	0	0	0	0
Mode 10	0	0	0	0	0	0	0	0	0	158	0	0	0	0	0	0	0
Mode 11	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0
Mode 12	0	0	0	0	0	0	0	0	0	0	0	42	0	0	0	0	0
Mode 13	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Mode 14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Unknown	0	1	0	0	2	2	2	9	5	2	0	3	0	0	0	0	0

Figure A.1: Confusion matrix of the test set using the Bilinear Eigenblades method.

	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6	Mode 7	Mode 8	Mode 9	Mode 10	Mode 11	Mode 12	Mode 13	Mode 14	Mode 15	Mode 16	Unknown
Mode 1	278	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 2	0	288	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 3	0	0	200	0	2	0	0	0	0	0	0	0	0	0	0	0	0
Mode 4	0	0	0	191	2	0	0	0	0	0	0	0	0	0	0	0	0
Mode 5	0	0	0	0	190	0	0	0	0	0	0	0	0	0	0	0	0
Mode 6	0	0	0	0	10	193	0	1	0	0	0	0	0	0	0	0	0
Mode 7	0	0	0	0	0	0	197	0	0	0	0	0	0	0	0	0	0
Mode 8	0	0	0	0	0	1	0	111	0	0	0	0	0	0	0	0	0
Mode 9	0	0	0	0	0	0	0	0	41	0	0	0	0	0	0	0	0
Mode 10	0	0	0	0	0	0	0	0	0	160	0	0	0	0	0	0	0
Mode 11	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0
Mode 12	0	0	0	0	0	0	0	0	0	0	0	44	0	0	0	0	0
Mode 13	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Mode 14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mode 16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Unknown	0	0	0	0	3	1	0	1	1	0	1	1	0	0	0	0	0

Figure A.2: Confusion matrix of the test set using the Triangle Interpolate Eigenblades method.

APPENDIX B. COMPLETE CODE LISTING

Listing 1 Adjusting the input file with the sample from Latin-hypercube sampling

```
def adjustInputFile(inputFile, doeArgs):
    # this function writes the template file prior to blade creation
    templateFile = "C:\Users\Alex\Desktop\ModeID\
    ModalAnalysis\InputFiles\inputFileTemplate.txt"
    with open(inputFile, 'w') as fileobj,
    open(templateFile, 'r') as templateFileObj:
        templateFileStuff = templateFileObj.readlines()
        SaveNum = 1
        numSecs = 4
        bChord = abs(doeArgs[0])
        bTwist = abs(doeArgs[1])
        bChordWOffset = abs(doeArgs[2])
        bFAOffset = abs(doeArgs[3])
        bHeight = abs(doeArgs[4])
        fileobj.write("SaveNum\t{}"
                      "\nnumSecs\t{}"
                      "\nbChord\t{}"
                      "\nbTwist\t{}"
                      "\nbChordWOffset\t{}"
                      "\nbFAOffset\t{}"
                      "\nbHeight\t{}\n".format(SaveNum, numSecs, bChord,
                      int(bTwist), bChordWOffset, bFAOffset, bHeight))
        fileobj.writelines(templateFileStuff)
```

Listing 2 Implementing Latin-hypercube sampling

```
def getDoESet(n=5, samples=10):
    # Latin-hypercube implmentation. n: the number of parameters,
    samples: the number of random values for parameter
    design = pyDOE.lhs(n, samples)
    means = [5.5, 2.5, 0.65, 0.65, 3.5]
    sdtvs = [.5, .5, .5, .5, .5]

    for index in xrange(len(means)):
        design[:, index] = norm(loc=means[index],
        scale=sdtvs[index]).ppf(design[:, index])

    return design
```

Listing 3 Executing the ANSYS APDL code from Python

```
def runANSYS(cwd):
    # executes the ANSYS command APDL file
    ansysEXE =
    '"C:\\Program Files\\ANSYS Inc\\v170\\ansys\\bin\\winx64\\ansys170.exe"'
    directionDir = '-p aa_t_a -dir "{}"'.format(cwd)
    runDir = '-j "displacement_run" -s read -l en-us -b'
    inputDir = '-i "C:\\Users\\Alex\\Desktop\\ModeID\\
    ModalAnalysis\\test.inp"'
    outputDir = '-o "{}\\file.out"'.format(cwd)
    outputStr = "{} {} {} {} {}".
    format(ansysEXE, directionDir, runDir, inputDir, outputDir)
    batchFile = "fullOptimizationRUNANSYS.bat"
    with open(batchFile, 'wb') as fileout:
        fileout.write(outputStr)

    p = Popen(batchFile)
    stdout, stderr = p.communicate()
```

Listing 4 Implementation of the design of experiments to create modal images.

```
import pyDOE
import os
from subprocess import Popen
import shutil
from scipy.stats.distributions import norm

def main():
    for masterIndex in range(1, 25): # where the blade images will run
        MasterDirectory = "C:\Users\Alex\Desktop\ModeMatch_Vision
        \ModeImages\\10_Mode_C_NACA7612_2\pictures{}".format(masterIndex)
        numberOfRuns = 10
        doe = getDoESet()  # call Latin-hypercube
        fileobj2 = open("C:\Users\Alex\Desktop\ModeMatch_Vision
        \ModeImages\\10_Mode_C_NACA7612_2\\mastercsv.txt", 'ab')
        for index in range(1, numberOfRuns+1):
            newDirectory = "{}\Run{}".format(MasterDirectory, index)
            if not os.path.exists(newDirectory):
                os.makedirs(newDirectory) # create the input file
                inputFile = "C:\Users\Alex\Desktop\ModeID\ModalAnalysis\
                InputFiles\inputFile.txt".format(index)
                adjustInputFile(inputFile, doeArgs=doe[index-1])
                # call bat file blade maker to make blade
                bladeMakerDir =
                "C:\Users\Alex\Desktop\ModeID\ModalAnalysis\\
                bladeCreator\MakeBlade.bat"
                bladePartDir = "C:\Users\Alex\Desktop\ModeID\
                ModalAnalysis\Blade1.prt".format(index)
                if os.path.isfile(bladePartDir):
                    os.remove(bladePartDir)
                p = Popen(bladeMakerDir)
                try:
                    stdout, stderr = p.communicate()
                except:
                    print "broken blade creator"
                    break
                runANSYS(newDirectory) # run ansys
                shutil.move(bladePartDir, newDirectory) # move part
                shutil.move(inputFile, newDirectory) # move inputFile
        fileobj2.close()
if __name__ == '__main__':
    main()
```

Listing 5 Implementation of the affine warp algorithm

```
import cv2
import numpy as np

def affine_warp_triangle(self, triangle_orig_coords, triangle_new_coords):
    img2 = 255 * np.ones(self.shape, dtype=self.dtype)
    # Define input and output triangles
    tri1 = np.float32(triangle_orig_coords)
    tri2 = np.float32(triangle_new_coords)
    # Find bounding box.
    r1 = list(cv2.boundingRect(tri1))
    r2 = list(cv2.boundingRect(tri2))
    # Offset points by left top corner of the
    # respective rectangles
    tri1Cropped = []
    tri2Cropped = []
    for i in xrange(0, 3):
        tri1Cropped.append(((tri1[i][0] - r1[0]), (tri1[i][1] - r1[1])))
        tri2Cropped.append(((tri2[i][0] - r2[0]), (tri2[i][1] - r2[1])))
    # Apply warpImage to small rectangular patches
    img1Cropped = self.img1[r1[1]:r1[1] + r1[3], r1[0]:r1[0] + r1[2]]
    # Given a pair of triangles, find the affine transform.
    warpMat = cv2.getAffineTransform(np.float32(tri1Cropped),
    np.float32(tri2Cropped))
    # Apply the Affine Transform just found to the src image
    img2Cropped = cv2.warpAffine(img1Cropped, warpMat, (r2[2], r2[3]),
    None, flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REFLECT_101)
    # Get mask by filling triangle
    mask = np.zeros((r2[3], r2[2], 3), dtype=np.float32)
    cv2.fillConvexPoly(mask, np.int32(tri2Cropped), (1.0, 1.0, 1.0), 16, 0)
    # Apply mask to cropped region
    img2Cropped = img2Cropped * mask
    # Copy triangular region of the rectangular patch to the output image
    img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]] =
    img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]] *
    ((1.0, 1.0, 1.0) - mask)
    img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]] =
    img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]] + img2Cropped
    return img2
```