Deep Learning the Dynamics of Mechanical Systems

Bram Wigmans

TU Delft

Deep Learning the Dynamics of Mechanical Systems

Thesis report

by

Bram Wigmans

to obtain the degree of Bachelor of Science
at the Delft University of Technology
to be defended publicly on October 20, 2023 at 15:00

Thesis committee:

Supervisors:	Dr. A. (Alexander) Heinlein
	Dr.ir. S. (Shobhit) Jain
External examiner:	dr. Y. (Yves) van Gennip
Place:	Faculty of Applied Mathematics, Delft
Project Duration:	March, 2023 - August, 2023
Student number:	4586093

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Faculty of Applied Mathematics · Delft University of Technology

Copyright © Bram Wigmans, 2023
All rights reserved.

Abstract

This paper examines whether complex high-dimensional data that describes the dynamics of a cantilever beam can be transformed into a less complex system. In particular, the transformation that is examined is the reduction of the dimension. An essential aspect of this study involves the implementation of a linear autoencoder, which is a type of machine learning model that possesses the capability to effectively reduce the dimensionality of input data while adeptly reconstructing the original dataset. The model performs well and is successful in reconstructing complex data via the less complex system. However, the model struggles if the dynamics are made more complex by adding an external force. Although the dynamics seem to be present in the results, the amplitudes differ.

Summary

This research investigates the possibility of representing the dynamics of a high-dimensional beam through a reduced set of differential equations in a lower-dimensional space. The high-dimensional beam dynamics are obtained by trajectory simulations using finite element software like COMSOL. To achieve this reduction, a machine learning model is implemented that can transform high-dimensional data into a two-dimensional latent space. The model that will be used is referred to as a linear autoencoder. The latent space that is obtained by encoding using the linear autoencoder is studied to discover the dynamics of the cantilever beam. Studying the dynamics of cantilever beams can be useful for the engineering of structural buildings or help in aircraft design.

The data that are used for the training and validation of the autoencoder are obtained by simulating the high-dimensional beam with as initial value the displacement that belongs to the first eigenfrequency obtained from eigenfrequency analysis in COMSOL. The linear autoencoder that has been trained on this data becomes successful in reconstructing the high-dimensional beam data using a two-dimensional latent space. This is evaluated by looking at the loss function and the decoded hidden layer compared to the original data.

The latent space is explored by constructing a system of differential equations. For this, the time derivative has to be calculated and this is done with finite difference methods of different orders. The system gets simulated and decoded back to evaluate it against the original input data. Also, the dynamics of the system are explored by comparing its eigenvalues to the eigenvalues of a damped oscillator system. The eigenfrequency, damping ratio, and angular frequency are closely approximated with small errors, the errors are influenced by different orders of finite difference methods.

To further explore the model, an external force is introduced to the cantilever beam. Without training a new model, an attempt is made to reproduce the effect of the external force by adding an encoded force to the lower-dimensional latent system. The obtained results show the same dynamics as the added external force to the high-dimensional system. However, it is observed that the amplitude of the results is consistently higher than the data from the cantilever beam.

Overall, the dynamics governed by a high-dimensional cantilever beam can be explored using a lower-dimensional system of differential equations. It has some trouble replicating the system if an external force is added due to numerical errors. However, for the system without external force, the model produces good results.

Contents

1 Introduction **1**

2 Background **2**
2.1 SINDy 2
2.2 Neural networks 3
2.3 Autoencoder 5
2.4 SINDy Autoencoder 6

3 Reproducing Results **8**
3.1 Pendulum Equation 8
3.1.1 Methodology 8
3.1.2 Results 10
3.2 Chaotic Lorenz Equation 13
3.2.1 Methodology 13
3.2.2 Results 14
3.3 Conclusion 16

4 Cantilever Beam Decoupled **17**
4.1 Constructing the Beam Data 17
4.2 Linear Autoencoder 20
4.2.1 Methodology 20
4.2.2 Results 21
4.2.3 Conclusion 23
4.3 Discovering Dynamics 23
4.3.1 Methodology 23
4.3.2 Results 25
4.3.3 Conclusion 28
4.4 Adding an External Force 28
4.4.1 Methodology 28
4.4.2 Results 29
4.4.3 Conclusion 32

5 Conclusion **33**

6 Discussion **34**

References **36**

A Appendix **37**
A.0.1 Similar Obtained Model 37
A.0.2 Matlab Code 38
A.0.3 Python Code 40

Nomenclature

List of Abbreviations

DOF	Degrees of Freedom
LHS	Left Hand Side
MSE	Mean Square Error
PCA	Principal Component Analysis
RHS	Right Hand Side
SINDy	Sparse Identification of Nonlinear Dynamics
SVD	Singular Value Decomposition

List of Symbols

$\mathbf{f}$	External Force
$\mathbf{X}$	Displacement Vector
$\mathbf{z}$	Latent Vector
$\dot{\mathbf{X}}$	Velocity Vector
$\mathcal{L}_{\frac{d\mathbf{x}}{dt}}$	SINDy X Loss Term
$\mathcal{L}_{\frac{d\mathbf{z}}{dt}}$	SINDy z Loss Term
$\mathcal{L}_{\text{recon}}$	Reconstructing Loss Term
$\mathcal{L}_{\text{reg}}$	Regularization Loss Term
ω	Angular Frequency
ϕ	Encoder
ψ	Decoder
ζ	Damping Ratio
C	Damping matrix
f	Eigen Frequency
K	Stiffness matrix
M	Mass matrix
t	Time(s)

1 Introduction

In engineering understanding the behavior of mechanical systems is crucial for a safe and efficient design. Recently deep learning has emerged as a powerful tool for analyzing complex data. Deep learning algorithms inspired by the human brain, possess the capacity to read through massive datasets, recognize interesting patterns, and reveal hidden correlations that might elude conventional analysis methods (LeCun et al., 2015). These algorithms can be applied to discover the dynamics of a mechanical system, including the movement of structural cantilever beams.

Cantilever beams are widely used in engineering applications serving as essential structural components for structural elements in buildings and bridges, such as balconies, machinery, and building tools like cranes. But it also extends to fields like aerospace engineering (Ghasemikaram et al., 2020) where cantilevered wing structures are essential components of aircraft design. Understanding the behavior of such beams ensures that airplanes can withstand the aerodynamic forces encountered during flight. The study of cantilever beam dynamics not only aids in predicting their behavior in specific scenarios but also offers the potential to unveil hidden patterns and correlations, shedding light on fundamental principles governing mechanical systems.

In this thesis, the dynamics of the cantilever beam are investigated. This is done using a machine learning approach. The dynamics of the cantilever beam can be investigated by simulating the behavior using finite element software packages like COMSOL Multiphysics®. The data that is produced is high-dimensional which makes it complicated to interpret. To make it easier to research the dynamics, a lower dimension/ less complicated system could be obtained that captures the same dynamics. The goal of this thesis is to learn such a lower dimensional system of differential equations governing the same dynamics as a cantilever beam using high-dimensional trajectory data from finite element simulations.

The structure of this thesis is as follows: in Chapter 2 literature research is done. Previous methods for discovering dynamics are studied and explained. The next chapter (Chapter 3) consists of reproducing results from a key research paper that has been studied in the first chapter. Chapter 4 will cover the development of a machine learning model. It will explain the data construction process of the cantilever beam simulations and introduce a linear autoencoder for transforming the high-dimensional beam to a lower dimension. The architecture of the model is explained, and its performance is evaluated. In the final part of this chapter, the problem will get more complex by adding external force to the cantilever beam and the machine learning model will be reevaluated. A conclusion has been provided in Chapter 5, and one can read about the discussion of this research in Chapter 6.

2 Background

In this chapter, previous literature is researched. Different methods exist for discovering the dynamics of differential equations. For this project, we are interested in machine-learning approaches. Below are various methods described, and finally, a combination is introduced.

2.1. SINDy

Sparse Identification of Nonlinear Dynamics or SINDy for short is a data-driven methodology used for discovering equations and underlying dynamics from observational data (Brunton et al., 2016). The key idea behind SINDy is to exploit the sparsity in many real-world systems. By assuming that the dynamics of a system can be represented by a combination of sparse functions, meaning that the combination consists of only a few terms, SINDy aims to identify the most relevant terms and their coefficients in this system of ordinary differential equations (ODEs), which can be first or second order. These coefficients are found by introducing a library of candidate functions and using regression or a least-square error algorithm to predict these terms.

Mathematically, one can represent a system of differential equations as Equation 2.1:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t)) \tag{2.1}$$

Where $\mathbf{x}\dot{(t)} \in \mathbb{R}$, the left-hand side (LHS), is the rate of change of the state of a system at a particular time and $\mathbf{f}(\mathbf{x}(t))$, the right-hand side (RHS), the active few terms that the SINDy algorithm wants to predict. These terms describe the dynamics of the system.

To determine the function f from data, we gather a time history of the state $\mathbf{X}$ and its derivative $\dot{\mathbf{X}}$. The data is collected at multiple time points: $t_1, t_2, \ldots, t_m$, and it is organized into two matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) & \mathbf{x}^T(t_2) & \cdots & \mathbf{x}^T(t_m) \end{bmatrix}$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^T(t_1) & \dot{\mathbf{x}}^T(t_2) & \cdots & \dot{\mathbf{x}}^T(t_m) \end{bmatrix}$$

These matrices represent the time history of the state and its derivative respectively.

The derivative of the snapshot data is not always known. A finite difference method can be used to approximate the derivative of time series data. By using the data points, one can approximate the derivative by examining the rate of change and applying finite difference methods. One example is the forward difference method:

$$\dot{\mathbf{x}}(t_i) \approx \frac{\mathbf{x}(t_{i+1}) - \mathbf{x}(t_i)}{t_{i+1} - t_i}$$

The $\mathbf{f}$ can be predicted by constructing a library of candidate functions $\Theta(\mathbf{X})$. For example, the polynomials sin(x) and e^x are common functions that are often used. Or $\mathbf{X}^{P_2}$ and $\mathbf{X}^{P_3}$ that are defined as the quadratic nonlinearities in the state $\mathbf{x}$. For example :

$$\mathbf{X}^{P_2} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1)x_2(t_1) & \dots & x_2^2(t_1) & \dots & x_n^2(t_1) \\ x_1^2(t_2) & x_1(t_2)x_2(t_2) & \dots & x_2^2(t_2) & \dots & x_n^2(t_2) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_1^2(t_m) & x_1(t_m)x_2(t_m) & \dots & x_2^2(t_m) & \dots & x_n^2(t_m) \end{bmatrix} \tag{2.2}$$

The candidate matrix can look like this:

$$\Theta(\mathbf{X}) = \Big[\mathbf{1} \quad |\mathbf{X}| \quad |\mathbf{X}^{P_2}| \quad |\mathbf{X}^{P_3}| \quad \dots \quad |\sin(\mathbf{X})| \quad |e^{\mathbf{X}}| \quad \dots\Big]^T$$

Since it is assumed that a few terms are active in each row of $\mathbf{f}$, we can view it as a sparse regression problem to determine which coefficients: $\Xi = \begin{bmatrix} \boldsymbol{\xi}_1 \\ \boldsymbol{\xi}_2 \\ \vdots \\ \boldsymbol{\xi}_n \end{bmatrix}$ are non-zero, indicating the terms we are seeking.

The overall problem is shown in Equation 2.3:

$$\dot{\boldsymbol{X}} = \Theta(\boldsymbol{X})\Xi. \tag{2.3}$$

To approximate the correct dynamics of a system, we need to determine Ξ. Often, linear regression is used, but to encourage sparsity, Lasso regression is employed to identify the few active terms. Lasso regression is a method used to perform linear regression with regularization. A common loss function for linear regression is the Mean Squared Error (MSE). For our problem without regularization, this looks like this:

$$\arg\min_{\Xi} \|\dot{\boldsymbol{X}} - \Theta(\boldsymbol{X})\Xi\|_2^2$$

By adding an L1 norm we promote sparsity by forcing some of the coefficients to zero. The L1 norm can be weighted as demonstrated by Cortiella et al. (2020) in their research. The loss function with the added Lasso sparsity penalty looks as follows:

$$\arg\min_{\Xi} \left(\|\dot{\boldsymbol{X}} - \Theta(\boldsymbol{X})\Xi\|_2^2 + \lambda\|\Xi\|_1\right)$$

Then, we can use regression techniques to calculate the terms.

2.2. Neural networks

Neural networks are a subset of machine learning, which is a component of artificial intelligence that enables machines to imitate human behavior (Jäger and Reisinger, 2022). AI systems handle complicated tasks similar to how humans solve problems. Machine learning algorithms are often classifiers that train a label based on data features. During training, the classifier is exposed to various data examples with features and their associated label. After this training phase, the label is not given, and now it predicts this label without being given the label explicitly using the previously mentioned training phase.

In this thesis, only unsupervised learning classifiers are of importance. This means there are no labels or classifications, instead, the algorithm looks for patterns or structures in unlabeled data. For the high-dimensional beam, the data used has only the displacement and velocity vector. For this problem, Neural networks and autoencoders can be useful.

A neural network is a machine-learning model that is inspired by the workings of the human brain (Han et al., 2018). The neural network consists of neurons that receive input data and generate output data. Similar to the human neuron, this does not happen at a constant rate, a certain threshold must be met.

The functions that take care of this requirement are activation functions. During the training phase, the activation functions between all these neurons are trained. From each neuron to the next neuron a weight is assigned and in each neuron, the received inputs are processed with a bias which is also trained during training. In Figure 2.1 one can see a typical architecture of a neural network

A neural network can mathematically be described as a directed acyclic graph consisting of multiple layers of the previously mentioned neurons. Typically, there are three types of layers: The input layer, the hidden layers, and the output layer. The input layer receives the initial data features. The hidden layers perform transformations on the input data through weighted combinations and activation functions. The output layer produces the final output of the network, which can be a single value (for regression) or a set of values.

Each layer consists of neurons that perform a weighted sum of its inputs, add a bias term, and, apply an activation function:

$$Y = f(\sum_{i=1}^{n} W_i X_i + b_i)$$

Where X_i is the outcome of the activation function of the weighted sum plus bias of a previous parent neuron. W_i represents the weight of the connections between the parent neuron and the current neuron. Y is the outcome of the current neuron. For a visual explanation see Figure 2.2.

f represents the activation function. Activation functions introduce non-linearity into the network allowing it to learn complex relationships in the data. Common activation functions include the sigmoid, ReLU (Rectified Linear Unit), and hyperbolic tangent (tanh) functions.

Neural networks are mostly used for supervised learning but there are uses for unsupervised learning as well, for example, an autoencoder.

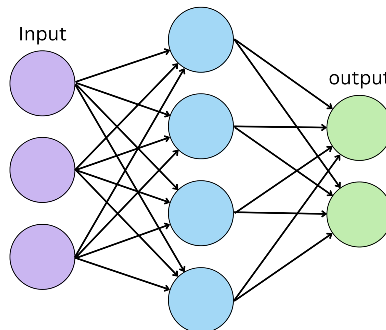


Figure 2.1: Neural network

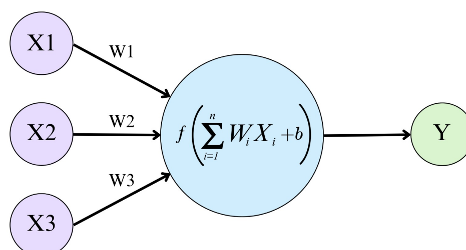


Figure 2.2: Visual Representation of Weight Function

2.3. Autoencoder

Autoencoders are a class of unsupervised Neural networks. Their primary function is to take input data and progressively transform it through a series of layers of neurons, ultimately reaching a central layer known as the latent layer. The latent layer is then connected to subsequent layers leading up to the output layer, which is designed to reconstruct the original input data. The layers between the input and output layers are called the hidden layers. Within the hidden layers, the dimensions of the data are typically reduced, which can result in some information loss. However, the neural network tries to minimize this information loss by capturing and retaining the essential features of the original input. It does this by training the weights of the activation functions between neurons. A simple visualization is shown in Figure 2.3, notice that the dimensions of the input and output layers are the same.

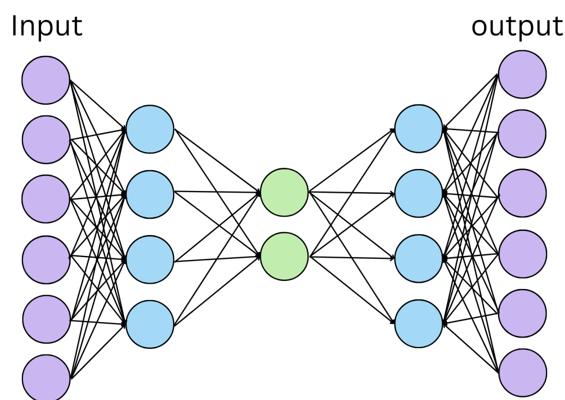


Figure 2.3: Visualization of an Autoencoder

To clarify, define the dimension of the input data as n and the dimension of the latent layer dimension as m. The transformation from the input layer to the latent layer is called the encoder, represented as a function $\phi : \mathbb{R}^n \to \mathbb{R}^m$. Conversely, the transformation from the latent layer back to the output data is called the decoder and is defined as $\psi : \mathbb{R}^m \to \mathbb{R}^n$. In this paper, we primarily focus on linear autoencoders, which do not employ activation functions. Instead, the whole network exists of linear functions connecting the neurons between the input and output layer. This implies that the network essentially acts as a linear map from its input layer to the output layer.

It's worth noting that there are other methods for reducing data to a lower-dimensional subspace using a linear map such as Principal Component Analysis (PCA). In Bourlard's work (Bourlard and Kamp, 1988) it was shown that a shallow undercomplete autoencoder (i.e., autoencoder with only one fully connected hidden layer), with linear output activation function and MSE cost function, learns weights that span the same subspace as the one spanned by principal component vectors (PCA). Moreover in the study by Plaut (Plaut, 2018) it is explained how one can obtain the PCA loading vectors that span this subspace, as they are not identical to the loading vectors of the autoencoder.

The autoencoder is trained by minimizing a loss function. The loss function for information loss can be custom-defined but commonly used is the mean square error:

$$\text{MSE}(x, \psi(\phi(x))) = \frac{1}{n} \sum_{i=1}^{n} \|x_i - \psi(\phi(x_i))\|^2$$

The way it minimizes this loss is during a training phase. During the training phase, the model employs an algorithm that optimizes the different weights such that the loss function is minimized. Most of the time this algorithm is based on the gradient descent.

Gradient descent works in the following way: Due to the chain rule, one can often calculate the derivatives of the loss function. By computing the derivatives we can identify local minima or maxima.

Over a certain amount of epochs or steps, we minimize the function. It begins with random weights for the encoder and decoder and calculates the derivative of the loss function. The algorithm then takes a step, often referred to as the learning rate, in the direction of the derivative in the weights of the neurons so that the derivative approaches zero. If the learning rate is chosen appropriately, the loss converges to a local minimum.

Autoencoders are valuable for addressing the research question since they reduce dimensionality while preserving important features, as mentioned earlier. The high-dimensional cantilever beam data can be encoded into a two-dimensional latent layer, which can be compared to a two-dimensional damped oscillating system.

2.4. SINDy Autoencoder

The SINDy autoencoder is a model that combines the SINDy algorithm with an autoencoder to capture the dynamics from scientific data. While SINDy can inherently capture dynamics, it requires the data to have an effective coordinate system for the dynamics to be represented simply (Champion et al., 2019). Often, scientific data lacks such a coordinate system. However, with the autoencoder, time series data can be transformed into a lower-dimensional latent layer, where an effective coordinate system may exist, allowing the SINDy algorithm to work effectively. To incorporate SINDy in the autoencoder and address the issue of the effective coordinate system in the latent layer, the loss function is modified by Champion et al. (2019).

The modified loss function is defined as:

$$\mathcal{L}_{\text{recon}} + \lambda_1 \mathcal{L}_{\frac{d\mathbf{x}}{dt}} + \lambda_2 \mathcal{L}_{\frac{d\mathbf{z}}{dt}} + \lambda_3 \mathcal{L}_{\text{reg}}, \tag{2.4}$$

Where the functions ϕ and ψ are the encoder and the decoder, as explained in the subsection above. The **x** is the input vector and **z** is a vector in the latent space.

If we have data with $\mathbf{X} = [\mathbf{x}_1\, \mathbf{x}_2\, \ldots\, \mathbf{x}_m]^T$ with $\mathbf{x} \in \mathbb{R}^n$

Defined is:

$$\boldsymbol{\Theta}(\mathbf{z}) = \begin{bmatrix} \theta_1(\mathbf{z}) \\ \vdots \\ \theta_p(\mathbf{z}) \end{bmatrix} \in \mathbb{R}^{m\times p}, \quad \boldsymbol{\Xi} = (\boldsymbol{\xi}_1\, \boldsymbol{\xi}_2\, \ldots\, \boldsymbol{\xi}_n) \in \mathbb{R}^{p\times n}$$

Here $\Theta_i(\mathbf{z})$ represents the p candidate functions for predicting the dynamics of the latent layer, and $\boldsymbol{\Xi}$ is the matrix containing the predicted coefficients for these candidate functions. By applying the chain rule, the following equations are obtained:

$$\dot{\mathbf{z}}(t) = \nabla_{\mathbf{x}}\phi\left(\mathbf{x}(t)\right)\dot{\mathbf{x}}(t), \quad \dot{\mathbf{x}}(t) = \nabla_{\mathbf{z}}\psi\left(\mathbf{z}(t)\right)\dot{\mathbf{z}}(t)$$

Loss term $\mathcal{L}_{\frac{d\mathbf{x}}{dt}}$

In the first loss term, the time derivative of the input data can be calculated using the latent layer since $(\nabla_{\mathbf{z}}\psi(\varphi(\mathbf{x}))) = (\nabla_{\mathbf{z}}\psi(\mathbf{z}))$ and $\left(\Theta(\varphi(\mathbf{x}))^T\Xi\right) = \dot{\mathbf{z}}$. It measures the loss in reconstructing the time derivative.

$$\mathcal{L}_{\frac{d\mathbf{x}}{dt}} = \left\|\dot{\mathbf{x}} - (\nabla_{\mathbf{z}}\psi(\varphi(\mathbf{x})))\left(\Theta(\varphi(\mathbf{x}))^T\Xi\right)\right\|_2^2.$$

In simple terms, the loss is defined as the comparison between the decoded time derivative of **z**, this derivative is calculated using the candidate functions and their coefficients, and the time derivative of **x**.

Loss term $\mathcal{L}_{\frac{d\mathbf{z}}{dt}}$
For the term of the loss of the time derivative of the latent vector. $\dot{\mathbf{z}}(t) = \nabla_{\mathbf{x}}\phi\left(\mathbf{x}(t)\right)\dot{\mathbf{x}}(t)$ and $\left(\Theta(\varphi(\mathbf{x}))^T\Xi\right) = \dot{\mathbf{z}}$. Thus, the loss function becomes:

$$\mathcal{L}_{\frac{d\mathbf{z}}{dt}} = \left\|\nabla_{\mathbf{x}}\phi(\mathbf{x})\dot{\mathbf{x}} - \mathbf{\Theta}(\phi(\mathbf{x}))^T\mathbf{\Xi}\right\|_2^2.$$

We encode the time derivative of **x** into the time derivative of **z** and compare it to the **z** derivative obtained from the candidate functions and coefficients.

Loss term $\mathcal{L}_{\text{recon}}$
The $\mathcal{L}_{\text{recon}}$ term is the standard loss in an autoencoder. It measures how well the autoencoder can reconstruct the data by encoding it and then decoding it.

$$\mathcal{L}_{\text{recon}} = \|\mathbf{x} - \psi(\varphi(\mathbf{x}))\|_2^2.$$

Loss term $\mathcal{L}_{\text{reg}}$
The final term $\mathcal{L}_{\text{reg}}$ is the term of the regularization of the coefficients of the candidate functions. To obtain sparsity, the L_1 norm gets evaluated as the loss $\mathcal{L}_{\text{reg}}$.

$$\mathcal{L}_{\text{reg}} = \|\mathbf{\Xi}\|_1$$

With the loss function designed, there is an autoencoder that uses SINDy to discover the dynamics of scientific data. The paper of Champion et al. (2019) gives various examples of real word problems. To research whether the dynamics of the beam can be captured using SINDy autoencoder, The results that Champion et al. (2019) have obtained are tried to be reproduced in the next chapter, (Chapter 3).

3 Reproducing Results

Since Champion et al. (2019) have put their code online, it is possible to attempt to reproduce the results mentioned in the previous chapter. Before this can be done, certain adjustments are necessary. The first adjustment involves updating TensorFlow, as the model uses TensorFlow 1 to predict the coefficients of a second or first-order differential equation. In the time between this research and the original model, TensorFlow 2 had been released, so the code has been modified to run TensorFlow 2. The adjustments primarily involved renaming variables, such as replacing “tf.” with “tf.compat.v1.”. The core architecture of the code remained unchanged.

The second adjustment is the storing of the loss. During the training phase of the SINDy autoencoder, the model of Champion et al. (2019) saved the validation loss at every 100 epochs. In order to reproduce and validate the results, the code has been modified so that the training loss and validation loss are stored at each individual epoch. The details can be found in the Appendix (A.0.3).

The systems of interest are the pendulum equation and the Lorenz chaotic system. The pendulum equation is of interest because this is described by a second-order differential equation and the high-dimensional cantilever beam is also described by a second-order differential equation. The chaotic Lorenz system is known for its unpredictability and complicated dynamics, making it interesting to see if the SINDy autoencoder can successfully replicate its behavior.

3.1. Pendulum Equation

3.1.1. Methodology

The pendulum equation 3.1 describes the motion of a pendulum that swings around a fixed point with an angle donated as ’z’ as shown in Figure 3.1. As mentioned by Champion et al. (2019), the data is a series of snapshot images with 51 by 51 pixels. Each image describes a 2D Gaussian centered at the pendulum’s center of mass, determined by the pendulum’s angle ’z’. Flattening these images results in the data denoted as $X(t) \in \mathbb{R}^{2601}$.

The training dataset is obtained by using 100 different trajectory data. For each time series, the trajectory data is obtained by Python ’odeint’ function, using the true model 3.1 and different initial conditions (representing the angle at which the trajectory begins at t=0). These trajectories start at ’t=0’ and extend to ’t=10’, with a time step of 0.02. This trajectory data is then converted into the previously mentioned image format. Each data entry thus represents an image with the location of the 2D Gaussian at a specific time. Consequently, the final training data set consists of 50000 images, obtained using 100 different time series. Different time points of the images of one such time series are shown in Figure 3.2.

The true model of the pendulum is shown in Equation 3.1:

$$\ddot{z} = -\sin(z) \tag{3.1}$$

In the paper by Champion et al. (2019), there are 10 models obtained by running the algorithm with the same parameters. Due to the neural network’s initialization with different values outside the hidden layer, there is an inherent randomness when the algorithm is run multiple times. In the paper’s appendix, it is

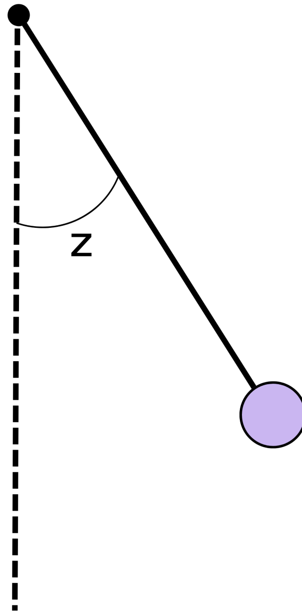


Figure 3.1: Swinging Pendulum

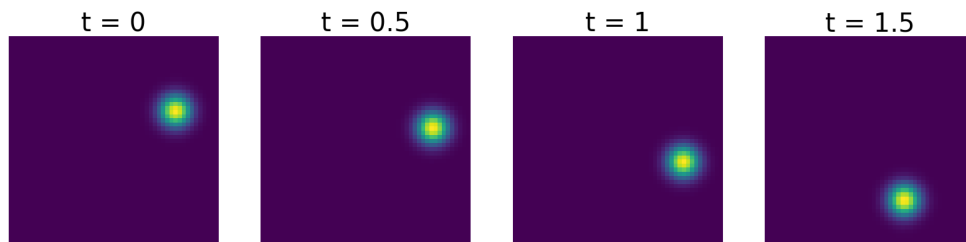


Figure 3.2: Images of one Time Series data.

noted that 5 of the 10 models predicted the true model correctly. However, the paper does not explicitly state the criteria for what qualifies as a 'correct' prediction, nor does it provide the exact 10 models obtained. instead, it states the model that has the least error and is closest to the true model, as represented by Equation 3.2:

$$\ddot{z} = -0.99\sin(z) \tag{3.2}$$

Additionally, there is a model that is not considered 'predicted correctly' despite sharing oscillatory properties with the true model, described by Equation 3.3.

$$\ddot{z} = -0.52z \tag{3.3}$$

As stated in the paper, “The four remaining models all have two active terms in the dynamics and have a worse performance than the models with one active term” (Champion et al., 2019). This might mean that the “correctly” predicted model may imply the selection of the correct candidate functions.

To replicate the results, the model is trained with the parameters mentioned in the article by Champion et al. (2019). The only difference is that the order of polynomials is now 1 instead of 3. This change is made

to investigate whether the model performs better when it has a higher chance to do so. To incorporate the parameters from the paper, the loss function for the pendulum equation has been adapted, resulting in the Equation 3.4:

$$\mathcal{L}_{\text{recon}} + 5 \times 10^{-4} \mathcal{L}_{\frac{d\mathbf{x}}{dt}} + 5 \times 10^{-5} \mathcal{L}_{\frac{d\mathbf{z}}{dt}} + 10^{-5} \mathcal{L}_{\text{reg}}, \tag{3.4}$$

Since $\sin(z)$ is enabled and the order of the single differential equation is 2, the candidate functions that the algorithm can choose from are : $1, z, \dot{z}, sin(z), sin(\dot{z})$ for f in

$$\ddot{z} = f(z)$$

The neural network has an input layer with 2061 neurons for $X(t) \in \mathbb{R}^{2601}$. There are three encoding layers with widths of 128, 64, and 32. Then, there is a latent layer with a dimension of 1. The decoding layer consists of 32, 62, and 128 neurons, and the final output layer has again, 2601 neurons. The activation function here is sigmoid, enabling the network to work with non-linear patterns in the data.

The network is trained for 6002 epochs as mentioned in the paper. In the first 5001 epochs, the candidate functions and coefficients are allowed to change freely. Every 500 epochs certain candidate functions are removed if their coefficient falls below 0.1 as mentioned in the paper, this process is referred to as 'sequential thresholding' and its purpose is to further promote sparsity. After 5000 epochs, the neural network fine-tunes the remaining coefficients of the candidate functions that are still present.

The running time of training this specific neural network takes approximately 8 hours. The extended running time is attributed to the need to calculate the challenging loss function for each epoch. The following models are obtained :

3.1.2. Results

Obtained model 1

$$\ddot{z} = -0.62 \sin(z) \tag{3.5}$$

In Equation 3.5 The SINDy autoencoder has identified the coefficient as 0.62 and the candidate function $\sin(z)$, as is shown in Equation 3.5. The autoencoder has correctly selected the candidate function, but the coefficient for the sin function has a value of -0.62 instead of -1. In Figure 3.3 one can see the model simulated next to the true model. Using 'odeint' with the same time parameters as before, and an initial z value of $\frac{\pi}{8}$, the simulated data from the obtained model seems to oscillate but at a different frequency.

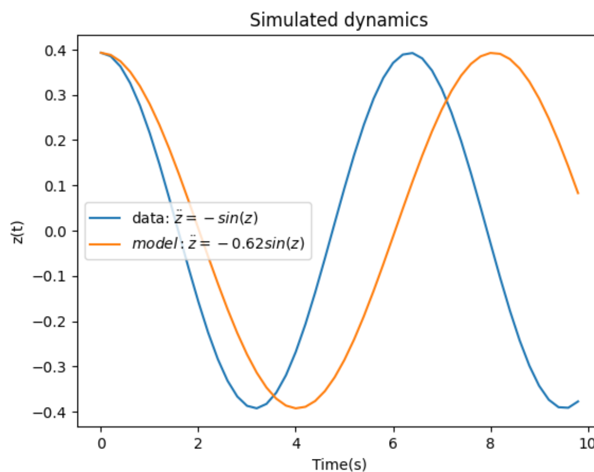


Figure 3.3

As one can see in Figure 3.4, the model thresholds three coefficients at 500 epochs. This number reduces at 2000 epochs, and finally, it halts at 3500 epochs, during which it refines the model to fit a

coefficient to the candidate function sin(z). The training loss and validation loss converge quite similarly until the later epoch, at which point the training loss surpasses the validation loss, as one would typically expect from a loss plot. This indicates a low probability of overfitting having occurred.

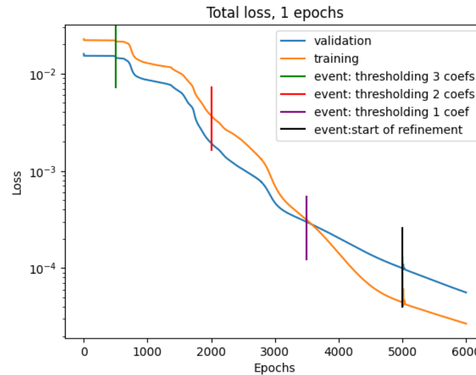


Figure 3.4: Plot of the Total loss of $-0.62\sin(z)$.

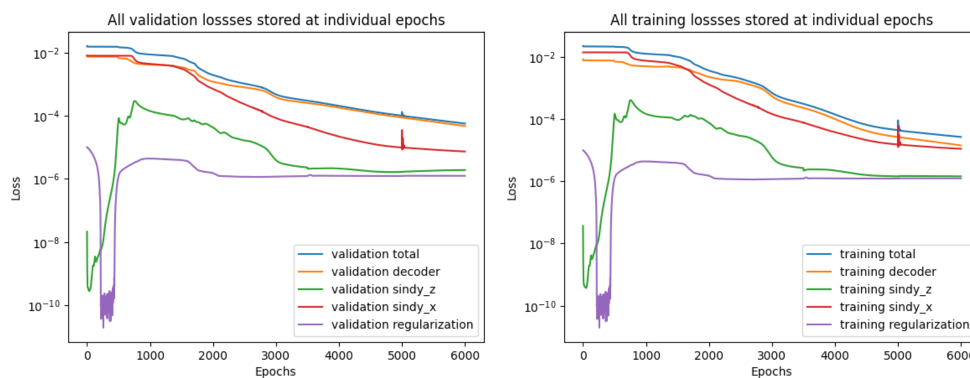


Figure 3.5: Individual Validation and Training Loss Plots of -0.62 sin(z)

In Figure 3.5 the individual losses for each loss term of Equation 3.4 are shown, for validation as well as for training the losses. The total loss is mostly attributed to the decoder loss term and the SINDy X loss term, defined as ($\mathcal{L}_{\frac{d\mathbf{x}}{dt}}$). The SINDy X loss as explained in the previous chapter is the loss term that ensures that the SINDy predictions can be used to reconstruct the original time derivatives of the input data. The other loss terms seem to not contribute much and stagnate after a certain amount of epochs.

Obtained model 2

$$\ddot{z} = -0.51z \tag{3.6}$$

The second model that the SINDy autoencoder has found is shown above in the form of Equation 3.6. The candidate function is now z with a coefficient of -0.51. One visible thing is that this is a result which is 1 decimal close to one of the models of the paper from Champion et al. (2019). Additionally, the simulation depicted in Figure 3.6 bears a resemblance to that in Figure 3.3. Again, sequential thresholding is enabled, as seen in Figure 3.7. It begins to threshold at epoch 500 in which it thresholds 2 coefficients and after that it thresholds 1 coefficient at 1500 epochs. In which the loss oscillates and eventually stagnates/converges to a certain loss. The validation loss ends lower than the training loss, which means once more that overfitting might not have happened.

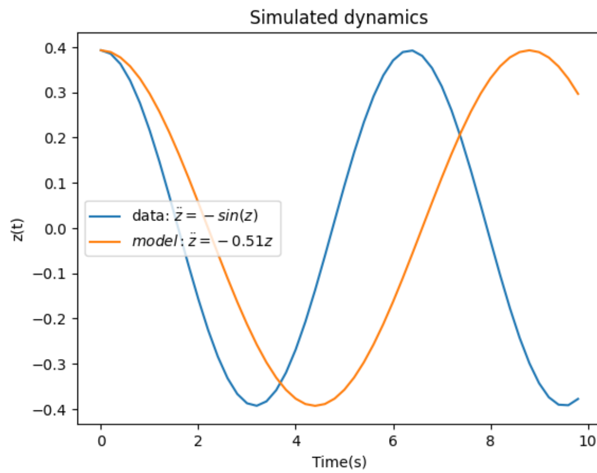


Figure 3.6

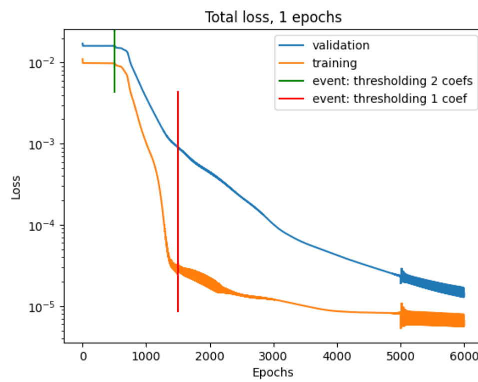


Figure 3.7: Plot of the Total Loss of $-0.52z$.

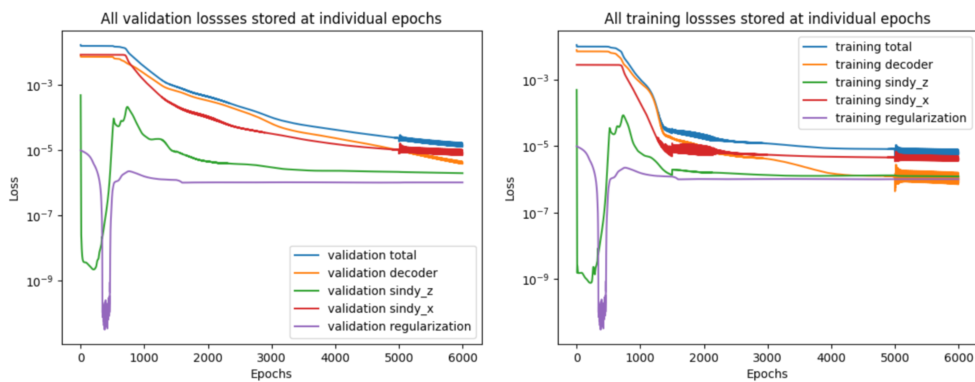


Figure 3.8: Individual Training and Validation Loss Plots of -0.51z

3.2. Chaotic Lorenz Equation

3.2.1. Methodology

The Lorenz system is a set of ordinary differential equations that exhibit chaotic behavior. It was first introduced by the mathematician Edward Lorenz in 1963 while studying atmospheric convection. The system is characterized by its sensitivity to initial conditions, leading to unpredictable and complex solutions. The set of ordinary equations is as follows:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

In the paper of Champion et al. (2019) the data that is being used for encoding is high dimensional data retrieved by choosing 6 fixed spatial nodes $\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}, \mathbf{u_5}, \mathbf{u_6} \in \mathbb{R}^{128}$ given by the first six Legendre polynomials with 128 gridpoints in domain [-1,1]. So $\mathbf{u_i} \in \mathbb{R}^{128}$ contains 128 points of the function $P_i(x)$ in which $-1 \leq x \leq 1$. With P_i, The Legendre polynomials, defined as:

$$P_0(x) = 1$$
$$P_1(x) = x$$
$$P_2(x) = \frac{1}{2}(3x^2 - 1)$$
$$P_3(x) = \frac{1}{2}(5x^3 - 3x)$$
$$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3)$$
$$P_5(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x)$$

Now, the input data for the input layer at a particular time is defined as:

$$\mathbf{x}(t) = \mathbf{u}_1 x(t) + \mathbf{u}_2 y(t) + \mathbf{u}_3 z(t) + \mathbf{u}_4 x(t)^3 + \mathbf{u}_5 y(t)^3 + \mathbf{u}_6 z(t)^3$$

$x(t), y(t), z(t)$ is trajectory data obtained using 'odeint' on the set of differential equations of the Lorenz system with parameters $\sigma = 10, \rho = 28, \beta = 8/3$. The goal of the autoencoder is to retrieve these trajectories and parameters in the latent layer from the newly formed $\mathbf{x}(t)$ and use SINDy to extract the correct candidate functions/coefficients.

The autoencoder parameters are the same as in the paper. The candidate functions do not include sin(z), so the candidate functions for each differential equation are all possible combinations of x, y, z including the constant function. The input layer has a dimension of 128, it is connected to an encoding layer with a dimension of 64 and this is connected to another encoding layer with a dimension of 32. Since our trajectory data has dimension 3, the latent layer has dimension 3. The two decoding layers have widths 64 and 128 which are connected to the output layer in which it ends with dimension 128. The loss function with parameters of the paper is described as the Equation 3.7.

$$\mathcal{L}_{\text{recon}} + \times 10^{-4} \mathcal{L}_{\frac{d\mathbf{x}}{dt}} + 0 \mathcal{L}_{\frac{d\mathbf{z}}{dt}} + 10^{-5} \mathcal{L}_{\text{reg}}, \tag{3.7}$$

3.2.2. Results

Obtained model 1

$$\begin{aligned} &\text{Predicted model} && \text{True model} \\ &\frac{dx}{dt} = -9.98x - 10y + 0.1xz && \frac{dx}{dt} = 10(y - x) \\ &\frac{dy}{dt} = -8.95xz - 0.809y && \frac{dy}{dt} = x(28 - z) - y \\ &\frac{dz}{dt} = 8.08 - 2.706z + 2.22xy && \frac{dz}{dt} = xy - 8/3z \end{aligned} \tag{3.8}$$

The model obtained from the SINDy autoencoder results differs from the true model as seen above in 3.8. In the predicted model, the first equation includes an additional term with a coefficient representing the function of xz, albeit barely meeting the threshold since it has a coefficient of 0.1. The coefficient for the term of the x direction is negative in comparison to the true model, which has a positive term. Which is the opposite direction in terms of dynamics

In the second equation, the coefficient for the y term is 0.809 which is close to 1. The candidate function that has been used in this equation is correct, but we are missing the candidate function for the x term. Additionally, the coefficient -8.95 is much lower compared to the value of 28.

In the third equation, we have all the correct candidate functions, except for the constant function. For the z term with a coefficient of 2.706, the value is very close to 2.665, but for the xy term the coefficient is twice as large as the true model. There is a second obtained model, but it is quite similar to the first obtained model. For this model, see the Appendix (A.0.1).

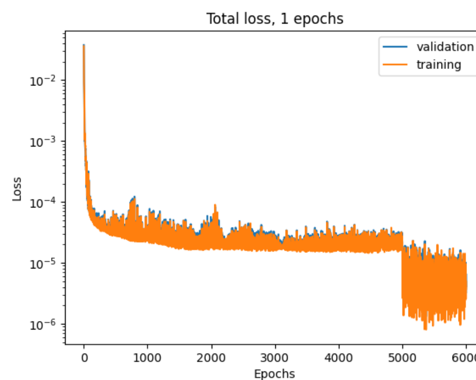


Figure 3.9: Plot of the Total Loss of the First Obtained Model

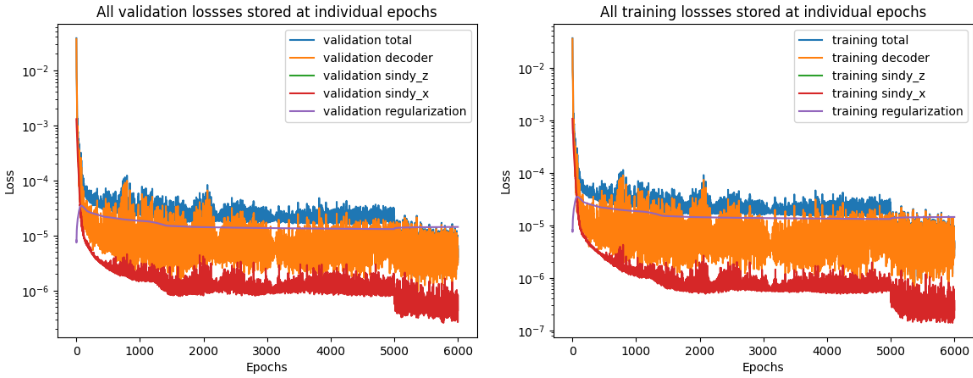


Figure 3.10: Individual Validation and Training Loss Plots of the First Obtained Model

3.3. Conclusion

In this section, the SINDy autoencoder results from the paper by Champion et al. (2019) were attempted to be reproduced. For the pendulum equation, the model predicted the correct candidate function one time out of two. The second predicted model is also an oscillating model, but not the correct candidate equation. Both models were simulated, and the simulation seems to oscillate like the true model, but not in the correct frequency. In the case of the Lorenz model, the SINDy autoencoder has attempted to identify the system's dynamics. While it has identified some promising candidate functions, there are some missing ones or, in some cases, an excessive number of functions. Additionally, the coefficients show significant variations.

In the paper by Champion et al. (2019), some models predict the right dynamics for a system of differential equations for the pendulum. Out of the ten models they have trained, only five of them successfully predict the correct model, which might correspond to identifying the right candidate function. The significant difference of -0.62 and -1 in the coefficients of the first model and the incorrect candidate function with the wrong frequency, give the reason to label them as incorrect. Reproducing the results for the paper might require more training with the same parameters, extensive fine-tuning of the four loss term parameters, or changing the widths or activation functions of the encoding or decoding layer, which can be time-consuming due to the long running time. The loss functions for the reproduced models converge, but this is mostly due to the decoder loss term, which might mean that the other terms are not useful. An autoencoder can encode and decode high-dimensional data, which is useful for researching the dynamics of a cantilever beam, but the difficult loss function that needs to be calculated each epoch costs too much time. It is expected that when the autoencoder encodes the high-dimensional beam data, the dynamics of the system's important features will be preserved in the lower-dimensional subspace, particularly considering the availability of velocity data for different element points.

In the upcoming chapter, we will tackle the issue of the high-dimensional beam by developing an autoencoder without the SINDy algorithm. By exploring the encoded latent dimension, we can investigate the dynamics and compare them to a two-dimensional damped oscillating system, thereby decoupling the problem.

4

Cantilever Beam Decoupled

In this chapter, the cantilever beam problem will be decoupled, separating the two tasks of dimensionality reduction and dynamics discovery. For dimensionality reduction, we employ autoencoders and construct training and testing datasets from unforced simulation trajectories, as discussed in Section 4.1. The architecture and evaluation of the autoencoder are detailed in Section 4.2. The dynamics of the hidden layer are explored through linear regression in Section 4.3. To test the capabilities of our learned reduced-order model, we attempt forced response predictions by introducing external forcing to the high-dimensional system. The autoencoder tries to predict the dynamics without prior training on these forced trajectories via the hidden layer. This is described in the final section of this chapter (Section 4.4).

4.1. Constructing the Beam Data

To construct the cantilever beam data, we utilize finite element solver COMSOL Multiphysics® as a black-box simulator. COMSOL Multiphysics® has a wrapping interface with the numerical linear algebra package MATLAB®.

The geometric object under consideration is a cantilever beam, meaning it is fixed at one end. The beam has 2 dimensions: the length and the thickness. The thickness of the beam measures 1 millimeter, and its length extends to 1 meter. The chosen material for the beam is structural steel since this material is often used in the real world. The material properties are shown in Table 4.1.

The finite element mesh consists of 1434 elements. The number of vertex elements is 4. The motion of the beam is restricted to 2D, and a total of 8602 degrees of freedom are solved in the mesh. The mesh's elements are classified into 1432 boundary elements and the remaining as interior elements. The mesh can be seen in Figure 4.1

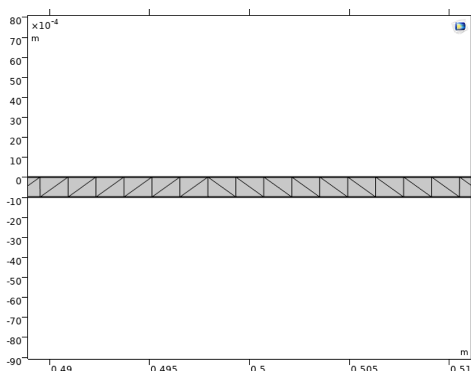


Figure 4.1: Visualization of Mesh

When the mesh is generated, the mass ($M \in \mathbb{R}^{8602\times 8602}$) and stiffness ($K \in \mathbb{R}^{8602\times 8602}$) matrices are generated. These matrices will be used to formulate the high-dimensional differential equation With $\mathbf{X}(\mathbf{t}) \in \mathbb{R}^{8602}$ representing the displacement vector with 8602 degrees of freedom (DOF).

Property	Symbol	Value	Units
Density	ρ	7850	kg/m^3
Young's modulus	E	200×10^9	Pa
Poisson's ratio	ν	0.30	1
Relative permeability	μ_r	1	1
Heat capacity at constant pressure	C_p	475	J/(kg·K)
Thermal conductivity	k	44.5	W/(m·K)
Electrical conductivity	σ	4.032×10^6	S/m
Relative permittivity	ϵ_r	1	1
Coefficient of thermal expansion	α	12.3×10^{-6}	1/K
Murnaghan third-order elastic moduli	l	-3.0×10^{11}	Pa
Murnaghan third-order elastic moduli	m	-6.2×10^{11}	Pa
Murnaghan third-order elastic moduli	n	-7.2×10^{11}	Pa
Lamé parameter λ	λ_{Lame}	1.15×10^{11}	Pa
Lamé parameter μ	μ_{Lame}	7.69×10^{10}	Pa

To compare the high-dimensional system of differential equations to a damped oscillator, damping is introduced to the system. The damping mechanism employed here is Rayleigh damping, achieved through a linear combination of the mass and stiffness matrices:

$$C = \alpha M + \beta K$$

The obtained C matrix is the damping matrix. To obtain enough damping α is set to 0.1 and β to 0.00001. β is a value close to zero to prioritize the mass proportional damping and to get less complex displacement vectors, meaning there is a smooth displacement vector with low values to the left at the fixed points and the highest values at the tip of the beam. The value for alpha is chosen such that there is enough movement in the beam in the time the model is simulated. For now, there is no external force, so $\mathbf{f} = \mathbf{0}$. The differential equation is shown in Equation 4.1.

$$M\ddot{\mathbf{X}} + C\dot{\mathbf{X}} + K\mathbf{X} = \mathbf{f}(t) \tag{4.1}$$

Eigenvalue Problem
To find one of the eigenmodes of the system of 4.1 needs to be transformed to an eigenvalue problem. We introduce a new variable, **Y**, defined as the first derivative of **X** with respect to time:

$$\mathbf{Y} = \dot{\mathbf{X}}$$

This transformation allows us to convert the system of second-order ODEs into a system of first-order ODEs:

$$\begin{cases} \dot{\mathbf{X}} = \mathbf{Y} \\ M\dot{\mathbf{Y}} = -K\mathbf{X} - C\dot{\mathbf{X}} \end{cases}$$

In matrix form, this system can be written as:

$$\begin{pmatrix} \dot{\mathbf{X}} \\ \dot{\mathbf{Y}} \end{pmatrix} = \begin{pmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \end{pmatrix}$$

where: - I is the identity matrix of appropriate size, - M^{-1} represents the inverse of the mass matrix M.

This system can be further condensed as $\dot{\mathbf{U}} = A\mathbf{U}$, where $\mathbf{U}$ is the vector $\begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \end{pmatrix}$, and A is the matrix

$$\begin{pmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{pmatrix}$$

.

The eigenvalue problem for this system is to find the eigenvalues λ and the corresponding eigenvectors $\mathbf{U}$ that satisfy:

$$(A - \lambda I)\mathbf{U} = 0$$

Solving this equation for λ yields the eigenvalues, which represent the natural frequencies of the system. The corresponding eigenvectors provide information about the mode shapes or vibrational patterns associated with each eigenfrequency.

COMSOL Multiphysics®

In COMSOL Multiphysics®, the eigenfrequency study on solid mechanics is performed to find the eigenfrequency, the eigenvalue/angular frequency, and the damping ratio of the different eigenmodes. We calculate these values around an eigenfrequency of 1 Hz to make sure there is enough movement in the system. The first eigenmode that is found describes the movement seen in Figure 4.2, and the values that are found are seen in Table 4.1. The movement described is the bending of the beam upwards and downwards in an oscillating movement at the middle of the beam.

Figure 4.2: Movement of the First Found Eigenmode

Variable	**Value**	Expression
Eigenfrequency	0.851+0.00798i Hz	f
Natural angular frequency/eigenvalue	5.3502+0.05016i Rad/s	$f2\pi$
Damping ratio	0.00937	Im$(f)/\lvert f\rvert$

Table 4.1: Variables and their Values

Using MATLAB®, the displacement vector corresponding to the described eigenmode is stored. This vector represents an eigenvector for the eigenvalue problem, which in this case, is 8602 DOFs. Since this is an eigenvalue problem, the trajectory data that will be retrieved corresponds to the eigenvector scaled in time. With the calculated eigenfrequency of 0.85Hz, the beam oscillates every 1.17 ($\frac{1}{0.85}$) seconds. The trajectory data would repeat itself indefinitely if there were no damping. However, when damping is introduced, the frequency remains constant while the amplitude decreases. The figure shown in 4.3 below displays the displacement data at the tip of the beam with damping.

For retrieving the data, the stored eigenvector $\mathbf{u}$ is scaled to $k\mathbf{u}$ with $k = 1, 2, \ldots, 10$. Scaling of the eigenvector can be described as bending the beam more and making the displacement higher. Then a

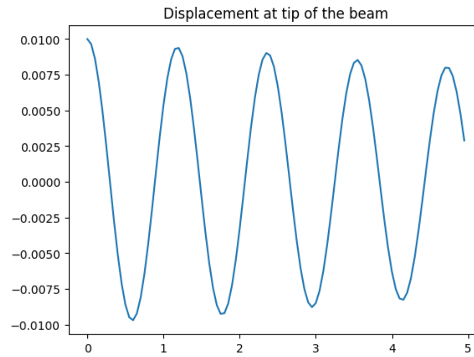


Figure 4.3

time transient study/simulation takes place with these scaled eigenvalues as initial values, and obtained are 10 different time series for the same model with different initial values. Since we are going to train a linear autoencoder, the time transient study is taking place without geometric non-linearity.

For the exact method on how the data is obtained, one can look at the MATLAB® code in the Appendix (A.0.2).

4.2. Linear Autoencoder

In this section, we present the architecture and results of a linear autoencoder employed for training. Since the problem has been decoupled, the SINDy component is no longer necessary. Given that our governing system in Equation 4.1 is linear, we consider dimensional reduction using a low-dimensional linear subspace of the high-dimensional state space containing displacement and velocity variables ($\mathbf{X}, \dot{\mathbf{X}}$). The primary objective of the autoencoder is to perform this linear transformation. The architecture for this autoencoder is described in the following subsection, and the subsequent subsection will evaluate the autoencoders performance.

4.2.1. Methodology

To train a linear autoencoder, it is essential to split the data into validation and training sets. Common practice is to allocate 80% of the data for training and the remaining 20% for validation. Therefore, out of the ten time series generated from ten different initial values, two will be reserved for validation. The data used for validation will be the ones generated with initial values $3\mathbf{u}$ and $7\mathbf{u}$. These scalar values are chosen since these two are not close together and one is on the lower end of the spectrum and the other one on the higher end.

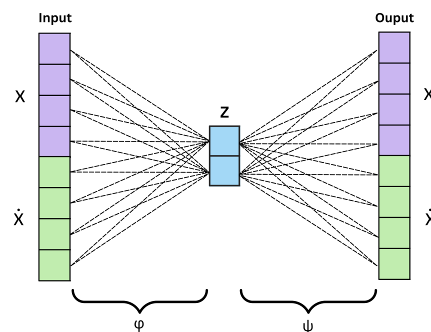


Figure 4.4: Visual Representation of the Architecture of the Linear Autoencoder

The linear autoencoder is constructed with a straightforward architecture, comprising an input layer, a hidden/latent layer, and an output layer. Both the input and output layers have a dimension of 17204, with the first 8602 dimensions representing the displacement vector and the remaining 8602 representing the velocity vector. The latent layer has a dimension of 2, matching the dimensionality of the oscillating damped

system. To capture the linear transformation, a linear activation function is used to propagate information across the different layers. Additional layers are unnecessary, as they would introduce non-linearity to our linear problem.

The transformation from the input layer to the hidden layer is defined as the encoder: $\phi : \mathbb{R}^{17204} \rightarrow \mathbb{R}^2$. The transformation from the latent layer to the output layer is defined as the decoder: $\psi : \mathbb{R}^2 \rightarrow \mathbb{R}^{17204}$. The loss function used during training is the Mean Squared Error (MSE).

The autoencoder is implemented using the Keras library from TensorFlow in Python, and a visual representation of it can be found in Figure 4.4.

4.2.2. Results

In Figure 4.5, it is clear that the loss of the linear autoencoder converges to a remarkably low value of approximately 10^{-14}. Throughout the training process, both the training and validation loss steadily decreased until they converged at around 150 epochs. The running time of training the linear autoencoder is 23 minutes and 30 seconds, including 5 minutes for loading the data into Python. Notably, the validation loss ends up lower than the training loss, but given the extremely low value, this discrepancy is negligible.

To evaluate the performance of the autoencoder, the hidden layers are visualized below with a plot (Figure 4.6) of the trajectory of the DOF with the max displacement, representing the tip of the beam. The latent space dimension is represented as z_1 and z_2. z_1 and z_2 both exhibit the same oscillatory pattern of the max displacement trajectory. This suggests that the autoencoder captured the important features of the input data.

Additionally, $\hat{\mathbf{x}} = \psi(\mathbf{z})$, which represents the hidden layer decoded, reconstructing the input data, has been plotted alongside the original input data in Figure 4.8. Once again, the DOF is chosen with the max displacement, representing the tip of the beam. A second plot is also provided alongside, which contains the velocity of this DOF, including both the original velocity data and the reconstructed data. The plots look identical, aligning with the observed loss patterns. The mean square error for reconstructing the displacement data at the tip is 6.77×10^{-12}, which is a significantly low value.

In Figure 4.7, the same plots are presented, but this time at a fixed point instead of the tip. Notably, the decoded data exhibits an oscillatory pattern, in contrast to the constant behavior of the input data. The plot is scaled in 10^{-13}, indicating that the decoded data has values close to zero.

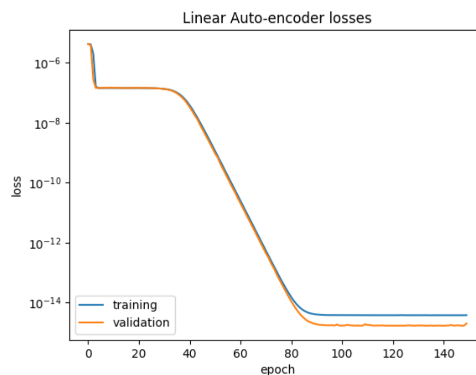


Figure 4.5: Training and validation loss of Linear Autoencoder

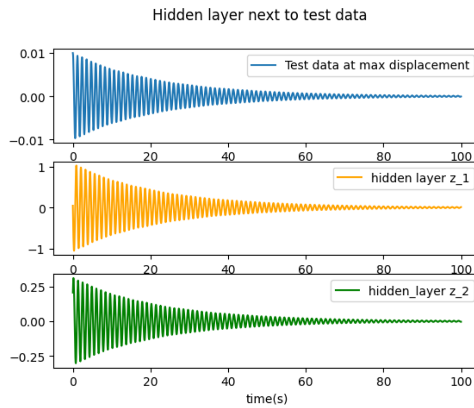


Figure 4.6: Comparison of the Decoded Hidden layer and the Test Data

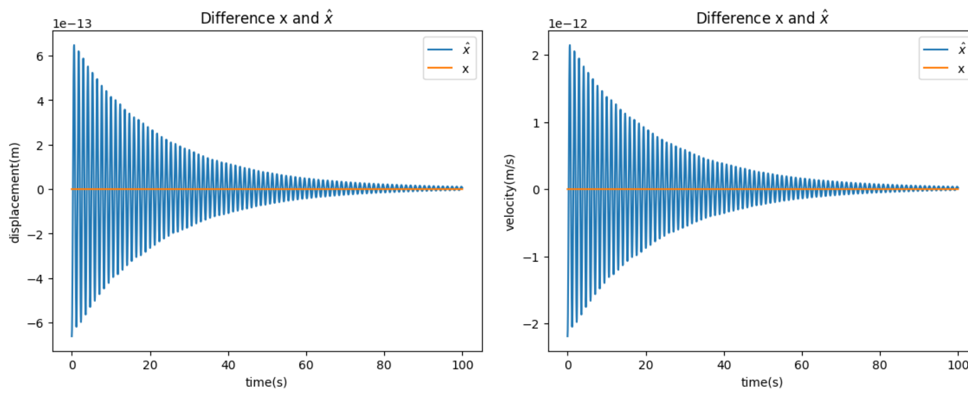


Figure 4.7: Comparison of the Decoded Hidden layer and the Test Data at a Fixed Point

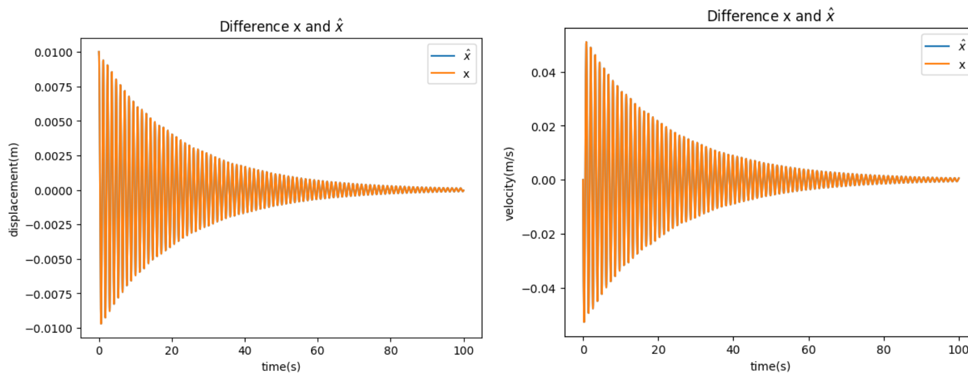


Figure 4.8: Comparison of The Decoded Hidden Layer and the Test Data at the Tip of the Beam

4.2.3. Conclusion

Overall, the linear autoencoder shows good performance in capturing the important features of a high-dimensional input vector into a lower-dimensional latent space. The convergence of the training and validation losses to a low value, along with the ability to capture the oscillations of the original data, together with the accurate reconstruction of the input data, shows that the linear autoencoder is effective in capturing and representing the underlying structure of the data. In the next section, we will study this hidden or encoded layer and compare this to the oscillating damped system.

4.3. Discovering Dynamics

With the promising linear autoencoder, the hidden layer can be compared to a damped harmonic oscillatory system. The differential equation governing the system is expressed in the Equation 4.2:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + 2\zeta\omega\frac{\mathrm{d}x}{\mathrm{d}t} + \omega^2 x = 0 \tag{4.2}$$

Here ω_0 is the harmonic frequency and ζ denotes the damping ratio. In the first subsection, the system of differential equations will be transformed into a system of first-order differential equations to calculate its eigenvalues. Subsequently, in the following subsection, the hidden layer will be explored by finding a system of differential equations and determining its eigenvalues. A comparison between the two sets of eigenvalues will provide insights into the relationship between the hidden layer and the damped harmonic oscillatory system.

this subsection will also involve simulations on the hidden layer. The decoded results obtained from these simulations will then be examined to determine if the high-dimensional beam data can be reconstructed using only the initial values of the encoded high-dimensional beam data.

4.3.1. Methodology

Deriving Eigenvalues

The differential equation shown in Equation 4.2 can be transformed to a 2-dimensional 1st order differential equation by introducing $\mathbf{v} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -w^2 & -2\zeta w \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Which can be represented as:

$$\dot{\mathbf{v}} = A\mathbf{v}$$

To find the eigenvalues of matrix A, we start by solving the characteristic equation. For a 2×2 matrix, the characteristic equation is given by:

$$\det(A - \lambda I) = 0$$

where I is the identity matrix. Substituting the values of matrix A and the identity matrix, we have:

$$\begin{vmatrix} -\lambda & 1 \\ -w^2 & -2\zeta w - \lambda \end{vmatrix} = 0$$

Expanding the determinant, we get:

$$(-\lambda)(-2\zeta w - \lambda) - (1)(-w^2) = 0$$

Simplifying and rearranging, we obtain the quadratic characteristic equation:

$$\lambda^2 + 2\zeta w\lambda + w^2 = 0$$

This equation can be solved using the quadratic formula:

$$\lambda = \frac{-2\zeta w \pm \sqrt{(2\zeta w)^2 - 4w^2}}{2}$$

Simplifying further:

$$\lambda = -\zeta w \pm w\sqrt{\zeta^2 - 1}$$

Since the damping ratio of the high-dimensional cantilever beam is lower than 1, there is an underdamped system. This means that the harmonic oscillatory damped system should be underdamped as well. The eigenvalues are therefore the complex conjugates:

$$\lambda_{1,2} = -\zeta w \pm iw\sqrt{1 - \zeta^2}$$

Comparing the Eigenvalues
To explore the dynamics of the hidden layer, we represent the hidden layer as a system with for the 2-dimensional vector $\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$ in Equation 4.3 :

$$\dot{\mathbf{z}} = B\mathbf{z} \tag{4.3}$$

In this subsection, the matrix B will be approximated using linear regression, and its eigenvalues will be calculated. $\dot{\mathbf{z}}$ will be approximated using various Finite Difference methods. The authors of (de Silva et al., 2020) created a Python package for the sparse identification of nonlinear dynamical systems from data. This pySINDy library provides a Python class called FiniteDifference, which calculates the derivative of input data and allows setting of a parameter defining the order of the finite difference method. We use central difference methods for even orders and a combination of central difference and Forward Euler for odd orders. In this subsection, order 1,2,3 will be considered to approximate the time derivative of the hidden layer. The above differential equation will then be studied to determine if it captures the dynamics of a damped oscillator.

If $\dot{\mathbf{z}}$ is approximated using the FiniteDifference class, the Matrix B can be calculated using linear regression. The Scikit-learn library provides a function for this purpose. With this B matrix, we can calculate the eigenvalues using the math library. Then the eigenfrequency, natural harmonic frequency, and the damping ratio can be calculated as follows:

If we set the eigenvalue which is complex equal to the previous calculated eigenvalues. we obtain:

$\lambda = a + bi = -\zeta w \pm iw\sqrt{1 - \zeta^2}$.

The real part is given by $a = -\zeta w$.

The imaginary part is $b = w\sqrt{1 - \zeta^2}$.

We can now derive the expression for $a^2 + b^2$:

$a^2 + b^2 = (-\zeta w)^2 + (w\sqrt{1 - \zeta^2})^2 = \zeta^2 w^2 + w^2 - w^2\zeta^2 = w^2$.

Thus, we find that $w = \sqrt{a^2 + b^2}$.

the damping ratio ζ can be determined by using the relationship $\zeta = \frac{a}{w} = Re(\lambda)/\,\|\lambda\|$. Finally, the eigenfrequency is the harmonic natural frequency w divided by 2π. All these variables are shown in 4.4.

$$\begin{aligned} w &= \sqrt{a^2 + b^2} \\ \zeta &= Re(\lambda)/\,\|\lambda\| \\ f &= \frac{w}{2\pi} \end{aligned} \tag{4.4}$$

4.3.2. Results

In Figure 4.9, the two-time derivatives and the regression coefficients c_1 and c_2 multiplied by the hidden layer are plotted. As one can see, the regression appears to be nearly identical to the approximated time derivatives. The coefficient of determination for the different orders is consistently around 0.99, indicating that the regression model almost perfectly fits the data. The coefficients are rounded for readability and can be seen below. Here B_i represents the B matrix obtained by using a Finite Difference method with order i $\in 1, 2, 3$.

$$B_1 = \begin{bmatrix} -11.78 & 7.25 \\ -20.57 & 10.28 \end{bmatrix} \quad B_2 = \begin{bmatrix} -11.10 & 7.27 \\ -20.62 & 11.01 \end{bmatrix} \quad B_3 = \begin{bmatrix} -11.24 & 7.35 \\ -20.86 & 11.13 \end{bmatrix}$$

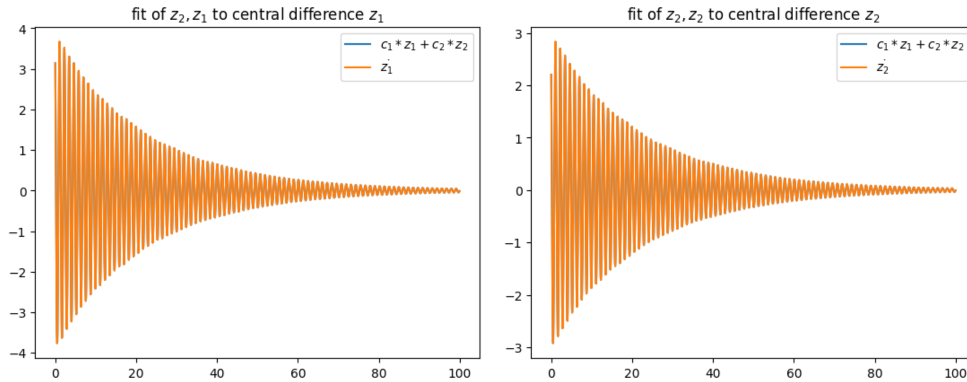


Figure 4.9: Regression of the Approximated Derivative with Order 1 of the Hidden Layer

The eigenfrequency, angular frequency, and damping ratio are then calculated using equations 4.4 and the eigenvalues of these matrices. the computed values are organized and presented in table 4.2.

Method	Eigenfrequency (f)	Angular Frequency (ω)	Damping Ratio (ζ)
True Model (COMSOL Multiphysics®)	0.85150	5.3501612663	0.009374926
Simple Forward Difference	0.8434776445623011	5.299726343248296	0.14105487430511587
Relative Error	0.0094268	0.0094268	14.0459715
Order 1	0.843477644406771	5.299726342271071	0.14105487068020153
Relative Error	0.0094268	0.0094268	14.0459715
Order 2	0.8369743862072992	5.258865165903355	0.008336535261038079
Relative Error	0.0177673	0.0177673	0.1353260
Order 3	0.8467579975751056	5.320337409100711	0.010086116345143871
Relative Error	0.0062757	0.0062757	0.0524727

Table 4.2: Comparison of Damping Ratio, Natural Frequency, and Eigenfrequency with True Model

Remarkably, The hidden layer successfully captures the essence of the harmonic damped model, as the calculated values are calculated with minimal error. However, it is noteworthy that The finite difference method with order 1 introduces computational errors, resulting in an inaccurate approximation of the damping ratio.

Simulations

From the different B matrices, the system can be simulated using 'odeint', a library for Python. The right-hand side corresponds to the different B matrices. The simulations are initialized with the initial conditions from the test data, which are encoded as the initial conditions for the hidden layer system.

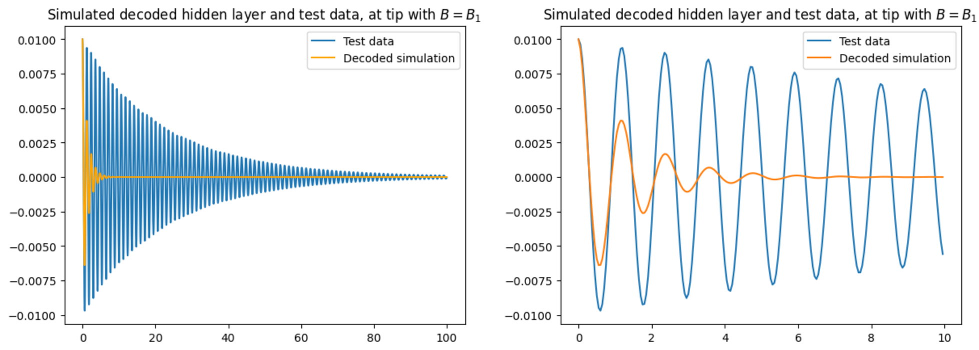


Figure 4.10: Decoded Simulation Compared to Test Data at the Tip, $B = B_1$

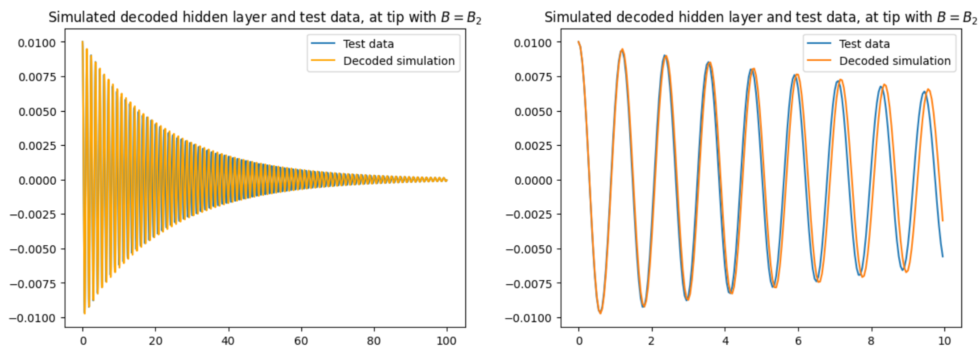


Figure 4.11: Decoded Simulation Compared to Test Data at the tip, $B = B_2$

In Figure 4.10, the decoded hidden layer and the test data initially overlap, but the simulation deviates from the correct path. This deviation can be explained by a large damping ratio of B_1 which leads to a rapid decrease in the simulation's amplitude, causing it to converge to 0 prematurely.

Figure 4.10 and 4.11 demonstrate that the simulation closely follows the test data. However, there are variations in the simulation results for B_2 and B_3 due to different damping ratios. The simulation of B_2 has a slightly higher amplitude, and its frequency is slightly off by a small margin. On the other hand, the simulation of B_3 has an almost identical frequency to the test data, but its amplitude is too large. The second set of plots (Figures 4.11, 4.12) display the first 10 seconds to examine these differences closely.

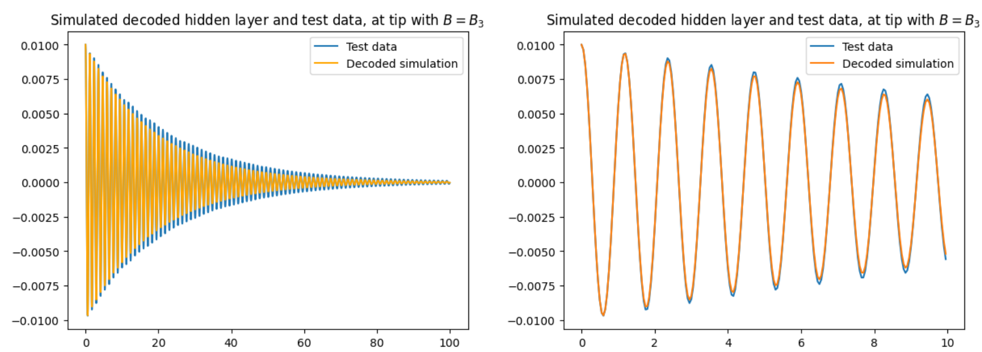


Figure 4.12: Decoded Simulation Compared to Test Data at the Tip, $B = B_3$

4.3.3. Conclusion

In this part of the study, we explored the dynamics of the hidden layer, representing it as a system of differential equations. The hidden layer was written in the form of $\mathbf{z} = \begin{bmatrix} z_1 \; z_2 \end{bmatrix}$, where $\dot{\mathbf{z}} = B\mathbf{z}$ and the unknown $\dot{\mathbf{z}}$ required numerical methods for approximation. Matrix B was approximated using linear regression and Finite Difference methods of different orders. The eigenvalues of these matrices were computed and compared to the eigenvalues of a damped harmonic oscillator. Using these eigenvalues, we determined the numerical approximations of the eigenfrequency, angular frequency, and damping ratio.

The hidden layer demonstrated remarkable accuracy in capturing the essence of the harmonic damped model, except for the system calculated using the first-order finite difference method. This system introduced some computational errors, leading to an inaccurate damping ratio approximation. For higher orders 2 and 3, the simulations of the different systems aligned with the test data, but there were slight differences in frequencies and amplitudes Nevertheless, the results for these systems show that the linear autoencoder can effectively capture the dynamics of a damped harmonic oscillator.

4.4. Adding an External Force

With the hidden layer successfully capturing a damped oscillating system, we can introduce more complexity by adding an external force to the tip of the high-dimensional beam. This leads to the following Equation 4.5:

$$M\ddot{\mathbf{X}} + C\dot{\mathbf{X}} + K\mathbf{X} = \mathbf{f}(t) \tag{4.5}$$

The force that is added is a zero vector except for the DOF representing the tip, where it is defined as $f_0 \cos(\Omega t)$:

$$\mathbf{f}(t) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ f_0 \cos(\Omega t) \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

In the following subsections, we will transform this system of equations into a system of equations that is expressed in values of the hidden layer. Since the high-dimensional data can be reconstructed using the hidden layer simulations, we can derive a lower-dimensional system with an external force by transforming **f** to a two-dimensional vector. With this system, new simulations can be made, and frequency response graphs can be generated, as shown in the results of this section. Analyzing these results will enable us to assess the autoencoder's performance in capturing the dynamics of this more complex system.

4.4.1. Methodology

The system described by Equation 4.5 can be transformed into a two-dimensional first-order system by introducing $\mathbf{Y} = \begin{bmatrix} \mathbf{X} \\ \dot{\mathbf{X}} \end{bmatrix}$ which can be represented as the Equation 4.6:

$$\dot{\mathbf{Y}} = A\mathbf{Y} + \begin{bmatrix} 0 \\ M^{-}1\mathbf{f}(t) \end{bmatrix}, \quad A = \begin{bmatrix} 0 & I \\ -M^{-1}K & M^{-1}C \end{bmatrix} \tag{4.6}$$

Now we can define **Y** as the decoded hidden layer $L\mathbf{z}$. The decoded function ψ can be described as a linear transformation L and the encoder function can be described as a linear transformation W. The

system 4.6 can be expressed as :

$$L\dot{\mathbf{z}} = ALz + \begin{bmatrix} \mathbf{0} \\ M^{-}1\mathbf{f}(t) \end{bmatrix}$$

$$WL\dot{\mathbf{z}} = WALz + W \begin{bmatrix} \mathbf{0} \\ M^{-1}\mathbf{f}(t) \end{bmatrix}$$

$$\dot{\mathbf{z}} = Bz + W \begin{bmatrix} \mathbf{0} \\ M^{-1}\mathbf{f}(t) \end{bmatrix}$$

We can compare this to the damped oscillatory system with external force:

$$\dot{\mathbf{z}} = Bz + \begin{bmatrix} f_{z_1} \cos \Omega t \\ f_{z_2} \cos \Omega t \end{bmatrix}$$

The force at the tip at $t = 0$ is f_0 since $\cos(0) = 1$. By applying a matrix multiplication with the inverse of M, which can be obtained from MATLAB® LiveLink™ (COMSOL, 2018), we encode this to obtain f_{z_1} and f_{z_2}.

$$\begin{bmatrix} f_{z_1} \\ f_{z_1} \end{bmatrix} = W \begin{bmatrix} \mathbf{0} \\ M^{-1}\mathbf{f}(t) \end{bmatrix}, \quad \mathbf{f}(t) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ f_0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

With the help of 'odeint', a library in Python that can simulate a differential equation, the differential equation of the hidden layer can be simulated and after decoding the trajectory of **z**, the frequency response can be calculated. The frequency response is the amplitude of the tip of the beam after the damping has stalled, and the remaining trajectory is that of the force at the tip. The system gets simulated for values around $\Omega = 5.35$ for $f_0 \cos(\Omega t)$ since this is the angular frequency where resonance is expected to occur, resulting in a peak in amplitude. The selected values are $\Omega = 4.5 + 0.025m$ with $m = 1, \ldots, 60$.

4.4.2. Results

In Figure 4.13, you can observe the frequency response plots of the decoded hidden layer and the tip of the high-dimensional beam with external force $\cos \Omega t$. The two plots appear similar, featuring a peak around 5.35, as expected. However, there is a distinct difference in scale, with the third plot highlighting this disparity. While the dynamics of the system are present, they differ in order of magnitude. In Figure 4.15 and 4.16, the decoded hidden layer and the data at the tip of the beam are displayed, again showing similarities in the data but differences in magnitude.

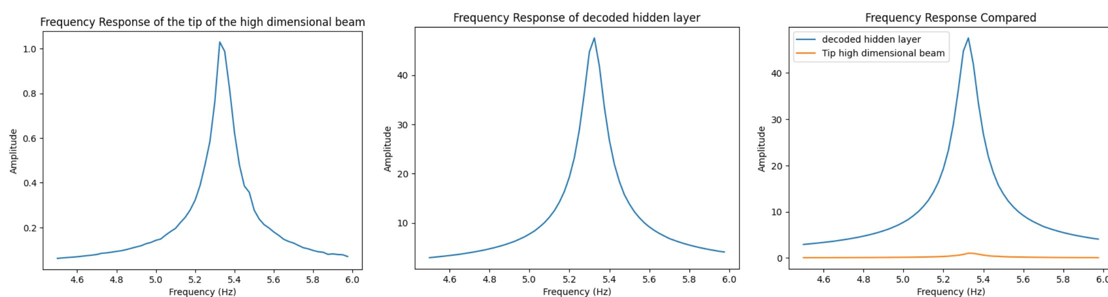


Figure 4.13: Frequency Response Plots of $\cos \Omega t$ External Force

The external force, now with an amplitude of 0.01: $0.01\cos\Omega t$ is also introduced to explore a different frequency response. As Figure 4.14 demonstrates, the frequency response exhibits a peak around the angular frequency, and the plots look similar, but the frequency response scale of the tip of the beam is a magnitude larger. In Figures 4.17 and 4.18 the trajectory data appears to exhibit the same behavior as the data obtained by applying an external force of $\cos\Omega t$.

The process of creating the high-dimensional data takes 2 hours and 8 minutes. Training the linear autoencoder requires 23 minutes and 30 seconds, including 5 minutes for reading the input data. The weights and architecture can be stored, so this step only has to run once. Running the simulation, decoding, and storing frequency responses takes 19 seconds. Obtaining the $M^{-1}\mathbf{f}$ vector in MATLAB® requires 1 minute. There is a significant decrease in running time because the simulations are now conducted in a low-dimensional latent space.

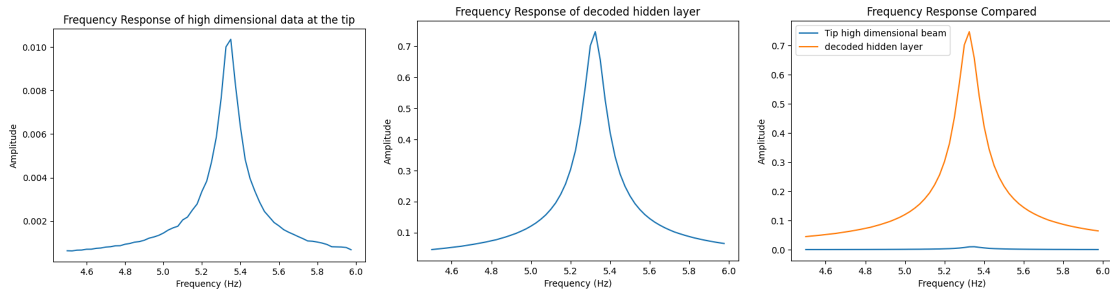


Figure 4.14: Frequency Response Plots of $0.01\cos\Omega t$ External Force

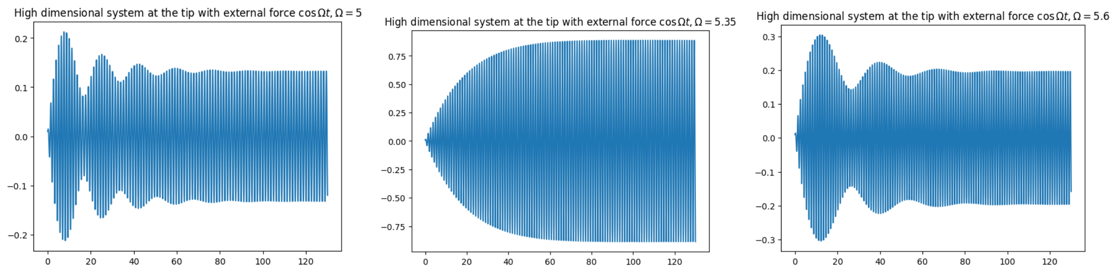


Figure 4.15: Trajectory Plots of Complex Data at the Tip With $\cos\Omega t$ External Force

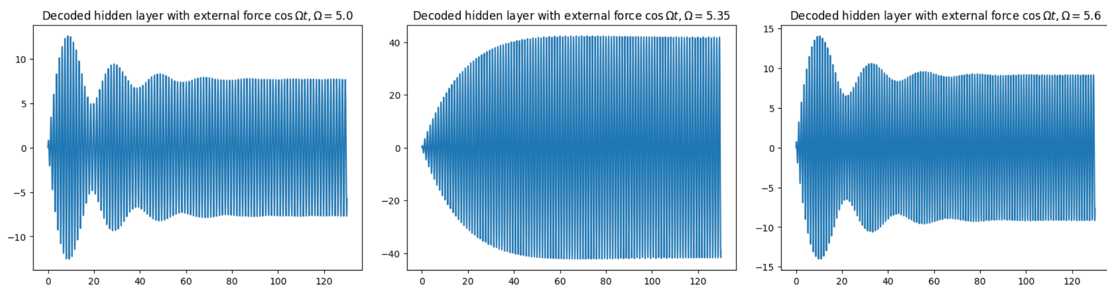


Figure 4.16: Trajectory Plots of Decoded Latent Layer Data at the Tip With $\cos\Omega t$ External Force

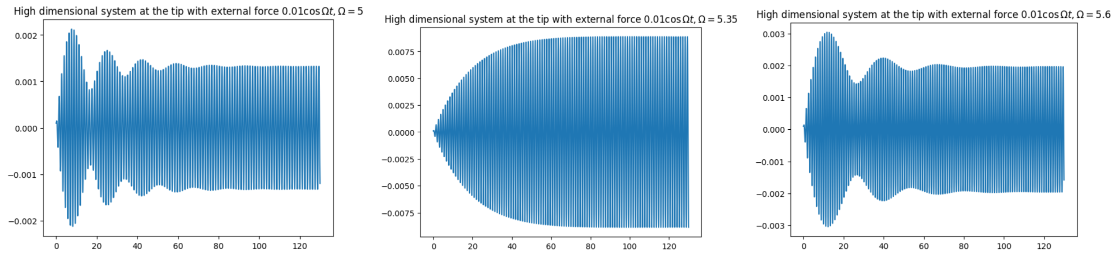


Figure 4.17: High Dimensional Data at the Tip Trajectory Plots With $0.01 \cos \Omega t$ External Force

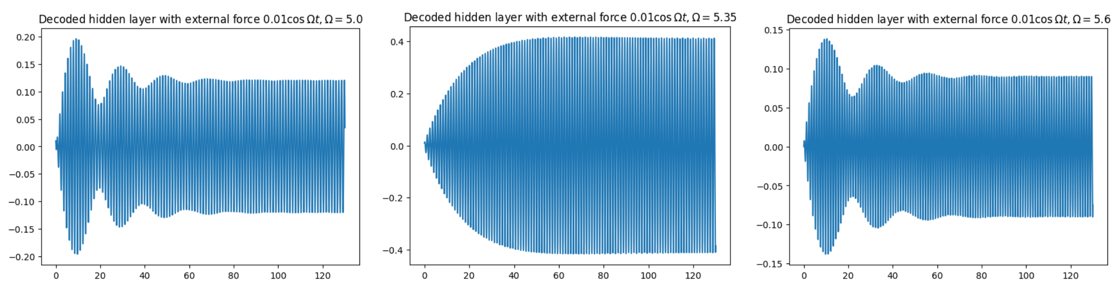


Figure 4.18: Decoded Hidden Trajectory Plots With $0.01 \cos \Omega t$ External Force

4.4.3. Conclusion

Introducing an external force to the system adds complexity. However, even with this increased complexity, it is possible to study the system's dynamics without directly using high-dimensional beam data subjected to an external force. Instead, the linear autoencoder is trained on a model without the external force, and the external force is then applied to the hidden layer. The behavior of external force appears similar and yields comparable results, but the true results differ in magnitude. This could be attributed to the computational errors in the matrix multiplication or the linear autoencoder not successfully predicting the correct decoder. Nevertheless, the linear autoencoder effectively captures the underlying dynamics of the system in a running time of 17 seconds, allowing us to gain valuable insights in a short time even in the presence of external forces.

5 Conclusion

This thesis researched the possibility of finding a lower-dimensional system of differential equations governing the high-dimensional dynamics using simulations from the finite element software package COMSOL Multiphysics®. To accomplish this, a linear autoencoder was implemented. The linear autoencoder demonstrates strong performance, enabling the reconstruction of trajectory data using a two-dimensional latent space. The total loss converges to a significantly low value, and the decoded trajectory data obtained by decoding the latent layer looks almost identical to the original input data. Notably, training the linear autoencoder requires less computational time than the SINDy autoencoder.

The dynamics of the latent layer govern an oscillating damped harmonic system with similar eigenfrequency, damping ratio, and angular frequency as the high-dimensional system of differential equations. The accuracy of these values depends upon the order of the finite difference method used to approximate the time derivative of the latent trajectory data. Since the dynamics of the latent layer govern an oscillating damped harmonic system, the linear autoencoder model can successfully capture the dynamics of a high-dimensional beam trajectory data without external force.

The previously trained linear autoencoder is less effective in reconstructing the trajectory data from the high-dimensional system when an external force is added. Attempting to reconstruct the high-dimensional data with external force without training data that includes such data appears challenging. This was intentionally done for research purposes. The reconstruction is done by encoding the force added in the high-dimensional system and applying this to the latent space, which behaves like an oscillatory-damped system. The simulation obtained with this system and added forces reproduces similar dynamics, but there is a magnitude difference in amplitudes. The primary reasons for this discrepancy are of a numerical nature.

Overall, the results show that the dynamics of a cantilever beam described in a high-dimensional system of differential equations can be captured by a machine learning model. It might not find the correct amplitudes when adding force. However, the behavior of the dynamics of the system with external force obtained from training without external force appears similar. Notably, the running time of obtaining the frequency response curves has significantly decreased. The systems that the linear autoencoder can capture are linear. The expectation is that non-linear dynamics can be captured, but the autoencoder has to be enhanced with different approaches. These approaches and reasons for the difference in amplitudes are discussed in the next chapter: the Discussion (Chapter 6).

6 Discussion

During this research, some problems were encountered. The main problem was the linear autoencoder not correctly predicting the amplitude of the external forced system. One of the reasons can be explained by looking at Equation 6.1, which describes the system of the beam without external force.

$$\dot{\mathbf{Y}} = A\mathbf{y}, \quad A = \begin{bmatrix} 0 & I \\ -M^{-1}K & M^{-1}C \end{bmatrix} \tag{6.1}$$

Since it is assumed that

$$\begin{aligned} L\dot{\mathbf{z}} &= ALz \\ WL\dot{\mathbf{z}} &= WALz \\ \dot{\mathbf{z}} &= WALz \end{aligned}$$

We can examine whether WLI_2 equals I_2 where I_2 is the identity matrix with dimensions 2 by 2. The following is obtained:

$$WLI_2 = \begin{bmatrix} 9.9999994 \times 10^{-1} & 2.4156179 \times 10^{-8} \\ -1.7520506 \times 10^{-7} & 9.9999982 \times 10^{-1} \end{bmatrix}, \quad I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

which is I_2 approximated. The second thing to inspect is whether $WAL = B$, with B the matrix that was obtained using the research in the system $\dot{\mathbf{z}} = B\mathbf{z}$.

$$B_3 = \begin{bmatrix} -11.24 & 7.35 \\ -20.86 & 11.13 \end{bmatrix} \quad , WAL(I_2) = \begin{bmatrix} -7.6848384 \times 10^{8} & 4.5505050 \times 10^{8} \\ -8.0029619 \times 10^{8} & 4.8082726 \times 10^{8} \end{bmatrix}$$

The matrix A has been obtained using MATLAB® with COMSOL Multiphysics®, along with the inverses of the large matrices M and K. The high values can be explained by the large numbers in the K matrix, where the maximum value is $max(k) = 1.151389599975440 \times 10^{12}$. The eigenvalues of WAL are real, indicating a potential issue with the calculation of WAL. Unfortunately, it is not possible to directly compare the B matrix to the theoretical WAL matrix.

MATLAB® with COMSOL Multiphysics® provides a function 'mphmatrix' for retrieving the matrices of the high-dimensional system of differential equations. Some of the matrices that can be obtained include $M, C, K, Null, Nullf, Mc, Cc, Kc$, with Mc, Cc, Kc the matrices with the DOF removed that are constrained (fixed points of the cantilever beam). These matrices can be computed by $Pc = Nullf^T P Null$. An interesting observation is that the Mc obtained with mphmatrix does not match the Mc calculated using the $Null$, $Nullf$, and M matrices. It's tempting to attribute this to the matrices provided by MATLAB®, but this remains an observation.

Another possible reason could be that the data intended for encoding does not fall within the correct range, as the autoencoder has been trained within a specific range of values. However, this is unlikely to be the case, as when the data is scaled to fit within the range, the outcome remains the same. Additionally, it's worth noting that it is a linear autoencoder, implying that the autoencoder operates as a linear transformation. The issue may be related to scaling in matrix B or the matrix M obtained from COMSOL Multiphysics®. In theory, the output should match the simulations performed in COMSOL Multiphysics®. Follow-up research using smaller matrices or displacement vectors with fewer degrees of freedom (DOF) could be interesting, as the linear autoencoder significantly reduces the running time of simulations of systems of differential equations with external forces.

The linear autoencoder employs linear activation functions, which are suitable for linear problems. To address the non-linearity introduced by altering the simulation settings in COMSOL Multiphysics®, the autoencoder must be enhanced by incorporating additional properties. One approach is to introduce more layers. When the input data gets more complicated, extracting features using a single layer becomes challenging. With the addition of more layers, each layer can capture essential features from the preceding one, reducing the risk of losing critical information. Another method to enhance the autoencoder's performance for more complex problems is to switch to non-linear activation functions. Linear activation functions are not proficient in learning non-linear relationships. By incorporating non-linear activation functions, the autoencoder becomes capable of capturing abstract features, making it more suitable for handling complex data. Researching the non-linear beam would be interesting follow-up research.

Another potential area for follow-up research is investigating different eigenmodes of the beam. In this research, the data used is derived from simulating using an initial value, which involves displacement by bending the beam in a specific manner. The displacement corresponds to the eigenvector associated with the first eigenfrequency, scaled to varying magnitudes. Additional eigenfrequencies with their respective eigenvectors and initial displacements could be computed using finite element software like COMSOL Multiphysics®.

References

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Ghasemikaram, A. H., Mazidi, A., Fazel, M. R., & Fazelzadeh, S. A. (2020). Flutter suppression of an aircraft wing with a flexibly mounted mass using magneto-rheological damper. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, *234*(3), 827–839. https://doi.org/10.1177/0954410019887039

Brunton, S. L., Proctor, J. L., & Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, *113*(15), 3932–3937. https://doi.org/10.1073/pnas.1517384113

Cortiella, A., Park, K.-C., & Doostan, A. (2020). Sparse identification of nonlinear dynamical systems via reweighted ℓ_1-regularized least squares. https://doi.org/10.1016/j.cma.2020.113620

Jäger, G., & Reisinger, D. (2022). Can we replicate real human behaviour using artificial neural networks? *Mathematical and Computer Modelling of Dynamical Systems*, *28*(1), 95–109. https://doi.org/10.1080/13873954.2022.2039717

Han, S.-H., Kim, K. W., Kim, S., & Youn, Y. C. (2018). Artificial neural network: Understanding the basic concepts without mathematics. *Dementia and neurocognitive disorders*, *17*, 83–89. https://doi.org/10.12779/dnd.2018.17.3.83

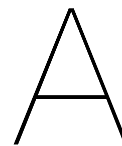
Bourlard, H., & Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, *59*, 291–294. https://doi.org/10.1007/BF00332918

Plaut, E. (2018). From principal subspaces to principal components with linear autoencoders.

Champion, K., Lusch, B., Kutz, J. N., & Brunton, S. L. (2019). Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, *116*, 22445–22451. https://doi.org/10.1073/pnas.1906995116

de Silva, B. M., Champion, K., Quade, M., Loiseau, J.-C., Kutz, J. N., & Brunton, S. L. (2020). Pysindy: A python package for the sparse identification of nonlinear dynamical systems from data. *Journal of Open Source Software*, *5*(49), 2104. https://doi.org/10.21105/joss.02104

COMSOL. (2018). *Livelink™ for matlab® user's guide*. https://doc.comsol.com/5.4/doc/com.comsol.help.llmatlab/LiveLinkForMATLABUsersGuide.pdf



Appendix

A.0.1. Similar Obtained Model

Predicted model	True model
$\frac{dx}{dt} = -0.8939x + 0.1xy - 8.778467xz$	$\frac{dx}{dt} = 10(y - x)$
$\frac{dy}{dt} = 8.19 - 2.68y + 2.22xz$	$\frac{dy}{dt} = x(28 - z) - y$
$\frac{dz}{dt} = -10.09y - 10.0176z$	$\frac{dz}{dt} = xy - 8/3z$

In the first equation, the predicted model only includes the correct candidate function x but with a coefficient that is not close to the true model. There is an additional variable z that is not seen in the corresponding equation of the true model.

In the second equation, the predicted model includes the constant function which is correct. However, it is missing the candidate function x that is present in the true model's second equation. Moreover, the sign of the coefficient of the xz term are in the opposite direction and differ in magnitude compared to the true model. The coefficient for the y term is three times larger than the corresponding coefficient in the true model.

The third equation has used one correct candidate function, namely the function z, but the value is significantly higher than the true model, suggesting a difference in the strength of influence for this term.

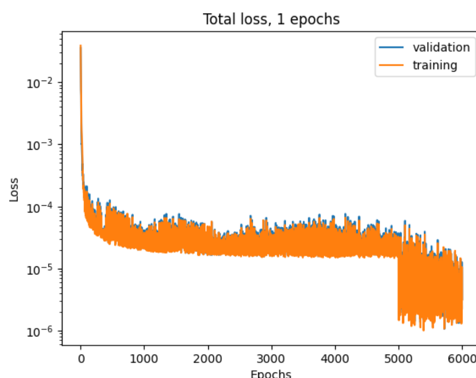


Figure A.1: Plot of the Total Loss of the Second Obtained Model

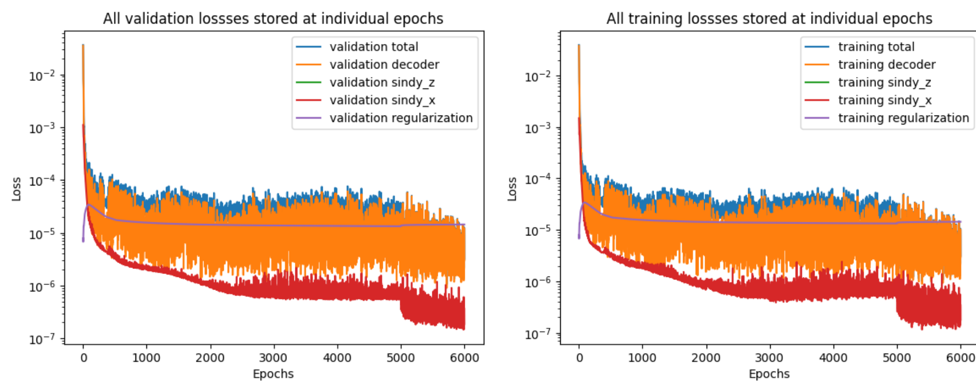


Figure A.2: Individual Validation and Training Loss plots of Second Obtained Model

A.0.2. Matlab Code

main.m

```
clear
import com.comsol.model.*
import com.comsol.model.util.*
clear;

still = transient_still()
MA_still = get_MA(still)

M_0 = MA_still.E;
C_0 = MA_still.D;
K_0 = MA_still.K;
Null = MA_still.Null;
Nullf = MA_still.Nullf;
ud = MA_still.ud;
uscale = MA_still.uscale ;



M = Nullf.' * M_0 * Null;

C = Nullf.' * C_0 * Null;

K = Nullf.' * K_0 * Null;


%NU = solinfo.sizesolvals;

M = MA_still.Ec;
data_u = cell(0);
data_udot = cell(0);

amount = 1


freq_resp = zeros(amount,1);

[model, u0, ud0] = get_ic()
```

```
idx_ones = u0==zeros(8602,1);
[max_num idx] = max(u0)
c = 0.001/max_num
u0 = u0*c
ud0 = ud0*c


f0 = 1
f = zeros(8596,1)
f(8594) = f0;

g = M\f
L1 = M\-K;
L2 = M\-C;

%Minv = inv(M)

for i = 1:1:amount
model = transient_still()
model.sol('sol1').setU(10*u0);
model.sol('sol1').setUDot(ud0);
model.sol('sol1').setPNames('t');
model.sol('sol1').setPVals(1);
model.sol('sol1').createSolution

model.study.create('std2');
model.study('std2').create('time', 'Transient');
model.study('std2').feature('time').set('tlist', 'range(0,0.1,5)');
model.study('std2').feature('time').set('probefreq', 'tout');
model.study('std2').feature('time').set('useinitsol', true);
model.study('std2').feature('time').set('initmethod', 'sol');
model.study('std2').feature('time').set('initstudy', 'std1');
model.study('std2').feature('time').set('solnum', 'first');
model.study('std2').feature('time').set('geometricNonlinearity', false);

model.sol.create('sol2');
model.sol('sol2').study('std2');
model.sol('sol2').attach('std2');
model.sol('sol2').create('st1', 'StudyStep');
model.sol('sol2').create('v1', 'Variables');
model.sol('sol2').create('t1', 'Time');

%model.study('std4').feature('time)
%0, 0.05, 100
%0, 0.  80
t_begin = 0;
t_end = 130;
timestep = 0.01;
model.sol('sol2').feature('t1').set('tlist', 'range('+ string(t_begin) +' ,' + string(tim
model.sol('sol2').feature('t1').set('timemethod', 'genalpha');
model.sol('sol2').feature('t1').set('tstepsgenalpha', 'strict');
model.component('comp1').physics('solid').create('pl1', 'PointLoad', 0);
model.component('comp1').physics('solid').feature('pl1').selection.set([4]);
%external_force = '0.01*cos(0.4+0.01*' + string(i) + '*t)'
%num = (i-1)*0.025+4.5;
num = 5.6;
```

```
numStr = num2str(num);
external_force = ['cos(t*' numStr ')']

%external_force = '0.01*cos(0.4+t*0.01)'

model.component('comp1').physics('solid').feature('pl1').set('Fp', {'0', external_force,

%model.sol('sol2').feature('t1').set('probefreq', 'tsteps');
%model.sol('sol2').feature('t1').set('atolglobalvaluemethod', 'factor');
model.sol('sol2').attach('std2')

model.study("std2").run();
t= t_begin:timestep:t_end;

[u udot] = mphgetu(model, 'soltag', 'sol2','solnum', 1:1:length(t));
data_u{i} = u;
data_udot{i} = udot;
freq_resp(i) = max(u(idx,2500:1:2601))
end
```

get_MA.m

```
function res = get_MA(model)

res = mphmatrix(model, 'sol1', ...
     'Out', {'K', 'D', 'E','Null', 'Nullf', 'ud', 'uscale', 'Ec', 'Dc' ,'Kc'}, ...
     'initmethod', 'init');



end
```

A.0.3. Python Code

SINDy autoencoder code change for storing loss

```python
def print_progress(sess, i, loss, losses, train_dict, validation_dict, x_norm,
                                                  sindy_predict_norm):
    """
    Print loss function values to keep track of the training progress.

    Arguments:
        sess - the tensorflow session
        i - the training iteration
        loss - tensorflow object representing the total loss function used in training
        losses - tuple of the individual losses that make up the total loss
        train_dict - feed dictionary of training data
        validation_dict - feed dictionary of validation data
        x_norm - float, the mean square value of the input
        sindy_predict_norm - float, the mean square value of the time derivatives of the input.
        Can be first or second order time derivatives depending on the model order.

    Returns:
        Tuple of losses calculated on the validation set.
    """
    training_loss_vals = sess.run((loss,) + tuple(losses.values()), feed_dict=train_dict)
    validation_loss_vals = sess.run((loss,) + tuple(losses.values()), feed_dict=validation_dict
                                                      )

    if(i%100 == 0):
        print("Epoch %d" % i)
        print("   training loss {0}, {1}".format(training_loss_vals[0],
                                               training_loss_vals[1:]))
```

```
        print("   validation loss {0}, {1}".format(validation_loss_vals[0],
                                          validation_loss_vals[1:]))
        decoder_losses = sess.run((losses['decoder'], losses['sindy_x']), feed_dict=
                                                  validation_dict)
        loss_ratios = (decoder_losses[0]/x_norm, decoder_losses[1]/sindy_predict_norm)
        print("decoder loss ratio: %f, decoder SINDy loss  ratio: %f" % loss_ratios)
    return training_loss_vals, validation_loss_vals
```

python linear autoencoder code

```
# %%
# Import necessary libraries
import os
import sys
import gc
import pandas as pd
import numpy as np
import tensorflow as tf
from keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Reshape, Flatten, Dense
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.python.keras import backend as K
from matplotlib import pyplot as plt
from mat4py import loadmat
import pysindy as ps

# Add the parent directory to the system path
sys.path.append('../../src')

# Import custom modules
from sindy_utils import sindy_fit
from sindy_utils import library_size
from training import train_network

# %%


# Disable GPU usage
os.environ["CUDA_VISIBLE_DEVICES"] = '-1'

# Check for available GPUs (if any)
available_gpus = K._get_available_gpus()

# Set the data path
data_path = os.getcwd() + '/'

# %%
# Load data from .mat files
u_data = np.array(loadmat(os.path.join(data_path, 'data', 'data_u.mat'))['data_u'])
udot_data = np.array(loadmat(os.path.join(data_path, 'data', 'data_udot.mat'))['data_udot'])


# %%
# Create dictionaries for training, validation, and test data
X_train = {}
X_val = {}
X_test = {}

# Create lists to store data
x_train = []
x_val = []
dx_train = []
dx_val = []
ddx_train=[]
ddx_val = []
dx_test = []
x_test = []

# Get the shape of the u_data array
```

```
shape_d = u_data.shape

# Iterate through the data and split it into train, validation, and test sets
for i in range(shape_d[0]):
    for j in range(shape_d[2]):
        if i == 3 or i == 7:
            x_val.append(u_data[i, :, j])
            dx_val.append(udot_data[i, :, j])
        elif i == 9:
            x_test.append(u_data[i, :, j])
            dx_test.append(udot_data[i, :, j])
        else:
            x_train.append(u_data[i, :, j])
            dx_train.append(udot_data[i, :, j])

# Store training data in the X_train dictionary
X_train['x'] = np.array(x_train)
X_train['dx'] = np.array(dx_train)

# Store validation data in the X_val dictionary
X_val['x'] = np.array(x_val)
X_val['dx'] = np.array(dx_val)

# Store test data in the X_test dictionary
X_test['x'] = np.array(x_test)
X_test['dx'] = np.array(dx_test)






# %%

# Define the encoding dimension
encoding_dim = 2

# Concatenate the training, validation, and test data
dX_train = np.concatenate((X_train['x'], X_train['dx']), axis=-1)
dX_val = np.concatenate((X_val['x'], X_val['dx']), axis=-1)
dX_test = np.concatenate((X_test['x'], X_test['dx']), axis=-1)

# Get the input dimension
input_dim = dX_train[0].shape[0]


# %%
# Create an array of time values
t = np.arange(0, 100.05, 0.05)

# Plot the data (displacement at the tip of the beam)
plt.plot(t[0:100], X_test['x'][0:100, 8599])
plt.title('Displacement at tip of the beam')
plt.show()

# %%
# Clear the Keras session
K.clear_session()

# Define the encoder input layer
encoder_input = Input(shape=(input_dim,))

# Encoder Layers (commented-out for reference)
# encoded1 = Dense(512, activation='relu')(encoder_input)
# encoded2 = Dense(256, activation='relu')(encoded1)
# encoded3 = Dense(256, activation='relu')(encoder_input)
# encoded4 = Dense(2250, activation='sigmoid')(encoded3)
# encoded5 = Dense(2000, activation='sigmoid')(encoded4)
# encoded6 = Dense(1750, activation='sigmoid')(encoded5)
# encoded7 = Dense(1500, activation='sigmoid')(encoded6)
# encoded8 = Dense(1250, activation='sigmoid')(encoded7)
```

```
# encoded9 = Dense(1000, activation='sigmoid')(encoded8)
# encoded10 = Dense(750, activation='sigmoid')(encoded9)
# encoded11 = Dense(500, activation='sigmoid')(encoded10)
# encoded12 = Dense(250, activation='sigmoid')(encoded11)
encoded13 = Dense(encoding_dim, activation='linear')(encoder_input)

# %%
# Decoder Layers (commented-out for reference)
# decoded1 = Dense(250, activation='sigmoid')(encoded13)
# decoded2 = Dense(500, activation='sigmoid')(decoded1)
# decoded3 = Dense(750, activation='sigmoid')(decoded2)
# decoded4 = Dense(1000, activation='sigmoid')(decoded3)
# decoded5 = Dense(1250, activation='sigmoid')(decoded4)
# decoded6 = Dense(1500, activation='sigmoid')(decoded5)
# decoded7 = Dense(1750, activation='sigmoid')(decoded6)
# decoded8 = Dense(2000, activation='sigmoid')(decoded7)
# decoded9 = Dense(2250, activation='sigmoid')(decoded8)
# decoded9 = Dense(256, activation='relu')(encoded13)
# decoded10 = Dense(256, activation='relu')(decoded9)
# decoded11 = Dense(512, activation='relu')(decoded10)

# Define the final decoder layer
decoded13 = Dense(input_dim, activation='linear')(encoded13)

# Create the autoencoder model
autoencoder = Model(inputs=encoder_input, outputs=decoded13)

# Print a summary of the autoencoder model
autoencoder.summary()


# %%
# Set random seeds for reproducibility
tf.keras.utils.set_random_seed(7)

# Enable op determinism for TensorFlow
tf.config.experimental.enable_op_determinism()

# Compile the autoencoder model
autoencoder.compile(optimizer='adam', loss=MeanSquaredError())

# Train the autoencoder model
history = autoencoder.fit(dX_train, dX_train, epochs=150, batch_size=100, verbose=1,
                                                validation_data=(dX_val, dX_val))

# %%
# Import necessary libraries
import matplotlib.pyplot as plt

# Plot the training and validation loss on a semilogarithmic scale
plt.semilogy(history.history['loss'], label='Training Loss')
plt.semilogy(history.history['val_loss'], label='Validation Loss')

# Print the last validation and training losses
print('Last Validation Loss:', history.history['val_loss'][-1])
print('Last Training Loss:', history.history['loss'][-1])

# Set the title and labels for the plot
plt.title('Linear Auto-encoder Losses')
plt.ylabel('Loss')
plt.xlabel('Epoch')

# Add a legend to the plot
plt.legend(loc='lower left')

# Show the plot
plt.show()


# %%
# Define the input layer for the encoded data
```

```
encoded_input = Input(shape=(encoding_dim,))

# Uncomment to create decoder layers manually (not recommended)
# decoded_layer_1 = autoencoder.layers[-4](encoded_input)
# decoded_layer_2 = autoencoder.layers[-3](decoded_layer_1)
# decoded_layer_3 = autoencoder.layers[-2](encoded_input)

# Create the final decoder layer by applying the last layer of the autoencoder to the encoded
                                                input
decoded_layer_4 = autoencoder.layers[-1](encoded_input)

# Create the decoder model
decoder = Model(encoded_input, decoded_layer_4)

# %%
# Create the encoder model
encoder = Model(inputs=encoder_input, outputs=encoded13)

# Encode the test data
encoded_x = encoder.predict(dX_test)

# Decode the encoded data using the decoder model
x_hat = decoder.predict(encoded_x)

# %%
# Create an array of time values
t = np.arange(0, 100.05, 0.05)

# Print the shape of the encoded data and decoded data
print("Shape of encoded data (encoded_x):", encoded_x.shape)
print("Shape of decoded data (x_hat):", x_hat.shape)

# Create subplots for the displacement of hidden layers and test data
fig, axs = plt.subplots(3)
fig.suptitle("Displacement of hidden layers next to test data")

# Plot the test data at max displacement
axs[0].plot(t[0:2001], X_test['x'][:, 8599], label='Test data at max displacement')
axs[0].legend()

# Plot the first hidden layer (z_1)
axs[1].plot(t[0:2001], encoded_x[:, 0], label='Hidden layer z_1', color='orange')
axs[1].legend()

# Plot the second hidden layer (z_2)
axs[2].plot(t[0:2001], encoded_x[:, 1], label='Hidden layer z_2', color='green')
axs[2].set_xlabel('Time (s)')
axs[2].legend()

# Create a new figure
plt.figure()

# Show the plots
plt.show()

# %%
import matplotlib.pyplot as plt

# Create a scatter plot of the encoded data (first 500 points)
plt.plot(encoded_x[0:500, 0], encoded_x[0:500, 1], marker='o', linestyle='', label='Encoded
                                                Data Points')

# Add labels and a legend
plt.xlabel('Hidden Layer 1')
plt.ylabel('Hidden Layer 2')
plt.title('Scatter Plot of Encoded Data')
plt.legend()

# Show the plot
plt.show()
```

```
# %%

# Create an array of time values
t = np.arange(0, 100.05, 0.05)

# Plot the difference between x and x_hat for the first dimension
plt.figure()
plt.plot(t[0:2001], x_hat[0:2001, 0], label=r'$\hat{x}$')
plt.plot(t[0:2001], dX_test[0:2001, 0], label='x')
plt.legend()
plt.ylabel('Displacement (m)')
plt.xlabel('Time (s)')
plt.title('Difference x and $\hat{x}$')

# Plot the difference between x and x_hat for dimension 8599
plt.figure()
plt.plot(t[0:2001], x_hat[0:2001, 8599], label=r'$\hat{x}$')
plt.plot(t[0:2001], dX_test[0:2001, 8599], label='x')
plt.legend()
plt.ylabel('Displacement (m)')
plt.xlabel('Time (s)')
plt.title('Difference x and $\hat{x}$')

# Calculate and print the sum of squared differences for dimension 0
print("Sum of squared differences (Dimension 0):", sum((x_hat[:, 0] - dX_test[:, 0]) ** 2))

# Plot the difference between x and x_hat for dimension 8602
plt.figure()
plt.plot(t[0:2001], x_hat[0:2001, 8602], label=r'$\hat{x}$')
plt.plot(t[0:2001], dX_test[0:2001, 8602], label='x')
plt.legend()
plt.ylabel('Velocity (m/s)')
plt.xlabel('Time (s)')
plt.title('Difference x and $\hat{x}$')

# Plot the difference between x and x_hat for dimensions 8599+8602
plt.figure()
plt.plot(t[0:2001], x_hat[0:2001, 8599 + 8602], label=r'$\hat{x}$')
plt.plot(t[0:2001], dX_test[0:2001, 8599 + 8602], label='x')
plt.legend()
plt.ylabel('Velocity (m/s)')
plt.xlabel('Time (s)')
plt.title('Difference x and $\hat{x}$')

# Show the plots
plt.show()


# %%
def calculate_velocity(array, delta_t, order):
    # Get the number of elements in the input array
    n = array.shape[0]

    # Initialize an empty list to store velocity values
    res = []

    if order == 2:
        # Add NaN values for the first and last elements (centered difference)
        res.append(float('nan'))
        for i in range(1, n - 1):
            vel_center = (array[i + 1] - array[i - 1]) / (2 * delta_t)
            res.append(vel_center)
        res.append(float('nan'))
    elif order == 1:
        # Calculate velocities using first-order forward difference
        for i in range(0, n - 1):
            value = (-1 * array[i] + array[i + 1]) / delta_t
            res.append(value)
        res.append(float('nan'))
    elif order == 3:
        # Calculate velocities using third-order finite difference approximation
```

```
        for i in range(0, n - 3):
            value = (-(11/6) * array[i] + 3 * array[i + 1] + (-3/2) * array[i + 2] + (1/3) *
                                                                    array[i + 3]) / delta_t
            res.append(value)
        res.append(float('nan'))
        res.append(float('nan'))
        res.append(float('nan'))
    else:
        return -1  # Invalid order

    # Return the calculated velocities as a numpy array
    return np.array(res)






# %%
import numpy as np
import math
from numpy import linalg as LA

def calculate_eigenvalue(coefs):
    # Convert the coefficients into a numpy array
    A = np.array(coefs)

    # Calculate eigenvalues and eigenvectors
    eigen_val, eigen_vec = LA.eig(A)

    # Extract the first eigenvalue
    ev1 = eigen_val[0]

    # Separate the real and imaginary parts
    a = ev1.real
    b = ev1.imag

    # Calculate the angular frequency (w) and damping ratio (d)
    w = -math.sqrt(a**2 + b**2)
    d = a / w

    # Calculate the natural frequency in Hertz
    natural_frequency_hz = abs(w) / (2 * math.pi)

    # Return the absolute values of eigenvalues, damping ratio, and natural frequency
    return abs(w), abs(d), natural_frequency_hz


# %%
import math
import numpy as np
from numpy import linalg as LA
from sklearn.linear_model import LinearRegression
import pysindy as ps
from pysindy.differentiation import FiniteDifference
import cmath

# Define an array of time values
t = np.arange(0, 100.05, 0.05)

# Extract the first and second components from the encoded data
z1 = encoded_x[:, 0].reshape(-1, 1)
z2 = encoded_x[:, 1].reshape(-1, 1)

# Create an empty array to store coefficients
lst = np.empty((len(z1), 2), dtype=float)

# Fill the lst array with z1 and z2 values
for i in range(len(z1)):
    lst[i, 0] = z1[i]
    lst[i, 1] = z2[i]
```

```
# Perform linear regression for order 1 finite difference
print("Simple forward difference")
vel_z1 = calculate_velocity(encoded_x[0:2001, 0], 0.05, 1).reshape(-1, 1)
vel_z2 = calculate_velocity(encoded_x[0:2001, 1], 0.05, 1).reshape(-1, 1)
reg_1 = LinearRegression(fit_intercept=False).fit(lst[0:2000], vel_z1[0:2000])
reg_2 = LinearRegression(fit_intercept=False).fit(lst[0:2000], vel_z2[0:2000])

c_central_z1 = reg_1.coef_[0]
c_central_z2 = reg_2.coef_[0]
print('z1 central c_1  =', c_central_z1[0])
print('z1 central c_2 =', c_central_z1[1])
print('z2 central c_1  =', c_central_z2[0])
print('z2 central c_2 =', c_central_z2[1])

# Calculate eigenvalues for the coefficients
coefs = [[c_central_z1[0], c_central_z1[1]], [c_central_z2[0], c_central_z2[1]]]
w, d, f = calculate_eigenvalue(coefs)
print('natural frequency:', w)
print('damping ratio:', d)
print('eigen frequency: ', f)

# Perform linear regression for order 1 finite difference using Python's FiniteDifference class
print("ORDER 1")
fd = FiniteDifference(drop_endpoints=True, order=1)
vel_z1_py = fd._differentiate(encoded_x[0:2001, 0], t=t[0:2001]).reshape(-1, 1)
vel_z2_py = fd._differentiate(encoded_x[0:2001, 1], t=t[0:2001]).reshape(-1, 1)

reg_3 = LinearRegression(fit_intercept=False).fit(lst[0:2000], vel_z1_py[0:2000])
reg_4 = LinearRegression(fit_intercept=False).fit(lst[0:2000], vel_z2_py[0:2000])

c_py_z1 = reg_3.coef_[0]
c_py_z2 = reg_4.coef_[0]

print('z1 py c_1 = ', c_py_z1[0])
print('z1 py c_2 = ', c_py_z1[1])
print('z2 py c_1 = ', c_py_z2[0])
print('z2 py c_2 = ', c_py_z2[1])

# Calculate eigenvalues for the coefficients
coefs1 = [[c_py_z1[0], c_py_z1[1]], [c_py_z2[0], c_py_z2[1]]]
w, d, f = calculate_eigenvalue(coefs1)
print('natural frequency:', w)
print('damping ratio:', d)
print('eigen frequency: ', f)


# %%
print("ORDER 2")
fd = FiniteDifference(drop_endpoints=True, order=2)

# Calculate second-order finite difference for z1 and z2
vel_z1_py = fd._differentiate(encoded_x[0:2001, 0], t=t[0:2001]).reshape(-1, 1)
vel_z2_py = fd._differentiate(encoded_x[0:2001, 1], t=t[0:2001]).reshape(-1, 1)

# Perform linear regression on the finite difference data
reg_3 = LinearRegression(fit_intercept=False).fit(lst[1:2000], vel_z1_py[1:2000])
reg_4 = LinearRegression(fit_intercept=False).fit(lst[1:2000], vel_z2_py[1:2000])

# Extract coefficients from the regression models
c_py_z1 = reg_3.coef_[0]
c_py_z2 = reg_4.coef_[0]

print('z1 py c_1 = ', c_py_z1[0])
print('z1 py c_2 = ', c_py_z1[1])
print('z2 py c_1 = ', c_py_z2[0])
print('z2 py c_2 = ', c_py_z2[1])

# Calculate eigenvalues for the coefficients
coefs2 = [[c_py_z1[0], c_py_z1[1]], [c_py_z2[0], c_py_z2[1]]]
w, d, f = calculate_eigenvalue(coefs2)
```

```
print('natural frequency:', w)
print('damping ratio:', d)
print('eigen frequency: ', f)

print("ORDER 3")
fd = FiniteDifference(drop_endpoints=True, order=3)

# Calculate third-order finite difference for z1 and z2
vel_z1_py = fd._differentiate(encoded_x[0:2001, 0], t=t[0:2001]).reshape(-1, 1)
vel_z2_py = fd._differentiate(encoded_x[0:2001, 1], t=t[0:2001]).reshape(-1, 1)

# Perform linear regression on the finite difference data
reg_3 = LinearRegression(fit_intercept=False).fit(lst[1:1999], vel_z1_py[1:1999])
reg_4 = LinearRegression(fit_intercept=False).fit(lst[1:1999], vel_z2_py[1:1999])

# Extract coefficients from the regression models
c_py_z1 = reg_3.coef_[0]
c_py_z2 = reg_4.coef_[0]

print('z1 py c_1 = ', c_py_z1[0])
print('z1 py c_2 = ', c_py_z1[1])
print('z2 py c_1 = ', c_py_z2[0])
print('z2 py c_2 = ', c_py_z2[1])

# Calculate eigenvalues for the coefficients
coefs3 = [[c_py_z1[0], c_py_z1[1]], [c_py_z2[0], c_py_z2[1]]]
w, d, f = calculate_eigenvalue(coefs3)
print('natural frequency:', w)
print('damping ratio:', d)
print('eigen frequency: ', f)

# %%
M = np.array(loadmat(data_path + '\data\M.mat')['M'])
C = np.array(loadmat(data_path + '\data\C.mat')['C'])
K = np.array(loadmat(data_path + '\data\K.mat')['K'])
L1 = np.array(loadmat(data_path + '\data\L1.mat')['L1'])
L2 = np.array(loadmat(data_path + '\data\L2.mat')['L2'])

I = np.identity(8596)
zer = np.zeros((8596, 8596))

A = np.block([[zer, I], [L1, L2]] )


# %%
g = np.array(loadmat(data_path + '\data\g001cos.mat')['g'])
g1 = np.array(loadmat(data_path + '\data\gcos.mat')['g'])
g2 = np.array(loadmat(data_path + '\data\gcos2.mat')['g2'])
g3 = np.array(loadmat(data_path + '\data\gcos3.mat')['g'])

# %%
print(g1[-1], g2[-1], g3[-1])

# %%
testje = np.identity(2)*1

testje = decoder.predict(testje)
encoder.predict(testje)

# %%
# Create an identity matrix of size 2
testje = np.identity(2)

# Use the decoder to predict the output of the identity matrix
p1 = decoder.predict(testje)

# Create an array to store the reshaped predictions
p = np.zeros((2, 8596 * 2))

# Reshape the predictions from p1 and concatenate them
for i in range(testje.shape[0]):
```

```
    l = np.concatenate([p1[i, 2:8], p1[i, 12:8602]])
    k = np.concatenate([p1[i, 8604:8610], p1[i, 8614:]])
    p[i, :] = np.concatenate([l, k])

# Calculate the result of A multiplied by the reshaped predictions
testje1 = A @ p.T

# Reshape the testje1 matrix
p2 = testje1.T
a = np.zeros((2, 8602))
b = np.zeros((2, 8602))
a[:, 2:8] = p2[:, 0:6]
a[:, 12:] = p2[:, 6:8596]
b[:, 2:8] = p2[:, 8596:8602]
b[:, 12:] = p2[:, 8602:]
p2 = np.concatenate([a, b], axis=1)

# Use the encoder to predict the output of p2
WAL = encoder.predict(p2)


# %%
# Calculate the eigenvalues of the matrix WAL
eigenvalues = LA.eig(WAL)[0]
print(eigenvalues)

# Calculate the natural frequency, damping ratio, and eigen frequency using the calculated
                                                 eigenvalues
w, d, f = calculate_eigenvalue(WAL)
print('natural frequency:', w)
print('damping ratio:', d)
print('eigen frequency: ', f)


# %%
from scipy.integrate import odeint
import numpy as np
from sklearn.metrics import mean_squared_error

# Convert coefficients to a numpy array
coefs = np.array(coefs3)

# Define time values
t = np.arange(0, 130.05, 0.05)

# Define a range of Omega values
Omega = np.arange(4.5, 6, 0.025)

# Define a function to describe the system dynamics
def func(z, t, args):
    z1, z2 = z
    return [
        (coefs[0][0] * z1 + coefs[0][1] * z2) + args[0] * np.cos(t * args[2]),
        (coefs[1][0] * z1 + coefs[1][1] * z2) + args[1] * np.cos(t * args[2])
    ]

# Initialize lists to store amplitudes
amps = []
amps_decoded = []

# Initialize vectors for calculations
tot = np.zeros((8602, 1))
tot[8599, :] = 1

a = np.zeros((8602, 1))
a[2:8] = g3[0:6]
a[12:] = g3[6:]

nal = np.zeros((8602, 1))
vec = np.concatenate((nal, a), axis=0)
```

```
# Predict using encoder
res = encoder.predict(vec.T)
arg1 = res[:, 0][0]
arg2 = res[:, 1][0]

j = 0
for omega in Omega:
    arg3 = omega
    args = [arg1, arg2, arg3]

    # Define initial conditions
    z0 = np.array([lst[0, 0], lst[0, 1]])

    # Integrate the system dynamics
    zz = odeint(func, z0, t, (args,))

    # Predict using decoder
    dec_zz = decoder.predict(np.array(zz))
    omega = round(omega, 2)

    # Store amplitudes
    amps.append([max(zz[2500:, 0]), max(zz[2500:, 1])])
    amps_decoded.append([max(dec_zz[2500:, 8599])])


# %%
from sklearn.preprocessing import normalize
import matplotlib.pyplot as plt

# Plot the frequency response of the decoded hidden layer
plt.figure()
# Calculate the L2 norm of amps_decoded
norm = np.linalg.norm(amps_decoded)
plt.plot(Omega, np.array(amps_decoded), label='Decoded hidden layer')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title("Frequency Response of Decoded Hidden Layer")

# Plot the frequency response of the tip's high-dimensional beam
plt.figure()
freq_resp = np.array(loadmat(data_path + '\data\/freq_resp2.mat')['freq_resp'])
plt.plot(Omega, freq_resp, label='Tip high-dimensional beam')
plt.plot(Omega, np.array(amps_decoded), label='Decoded hidden layer')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title("Frequency Response Comparison")
plt.legend()

# Plot the frequency response of high-dimensional data at the tip
plt.figure()
plt.plot(Omega, freq_resp, label='High-dimensional data at the tip')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title("Frequency Response of High-Dimensional Data at the Tip")
```