Reduced Order Modelling using Dynamic Mode Decomposition and Koopman Spectral Analysis with Deep Learning

David Krantz & Olof Månsson

Master's thesis
2020:E45

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Reduced Order Modelling using Dynamic Mode Decomposition and Koopman Spectral Analysis with Deep Learning

David Krantz
tfy15dkr@student.lu.se

Olof Månsson
tfy15oma@student.lu.se

June 23, 2020

Master's thesis work carried out at Haldex Brake Products AB

Supervisors:
Philipp Birken, philipp.birken@na.lu.se
Pontus Fyhr, pontus.fyhr@haldex.com

Examiner:
Tony Stillfjord, tony.stillfjord@math.lth.se

Abstract

In the industry simulation models are commonly used in system development. These models can become complicated in order to capture the physical behaviour of the underlying dynamical system. A high-fidelity representation, which can result in long simulation times, is in some settings not strictly required. A method for overall model fidelity reduction is therefore of interest.

In this thesis, two data-driven methods for model order reduction based on modal decomposition of data have been investigated. More specifically, we explore dynamic mode decomposition (DMD) and Koopman spectral analysis with deep learning. We validated this approach by extracting dominant dynamical characteristics from data for reduced order modelling. This was achieved by applying the methods to both linear and non-linear dynamical systems.

A reduced order model of a large highly non-linear dynamical system with meaningful fidelity was produced using the DMD method. Additionally, this reduced order model can be simulated significantly faster than the original high-fidelity model. The Koopman method was successfully applied to smaller non-linear systems, but did not capture the dynamics of the large highly non-linear system.

Keywords: non-linear dynamical systems, modal decomposition, Koopman operator theory, machine learning, data-driven analysis

Acknowledgements

We would like to thank our supervisor at Haldex, Pontus Fyhr, for his continuous support and enthusiasm. We would also like to thank our supervisor at LTH, associate professor Philipp Birken for guidance and great advice throughout our thesis. Lastly, we would like to express our gratitude to Haldex for giving us this opportunity and providing us with the necessary tools for this project.

Contents

Chapter 1

Introduction

Differential equations can be used to describe the world around us, from swinging pendulums to the interaction between nucleons. The area of numerical analysis revolves e.g. around the development of algorithms that can be used to find approximations to solutions to such equations. Advances in numerical algorithms and computing power have led to an increasing fidelity of numerical simulations. These simulations however often suffer from long computational times and may have higher fidelity than strictly required for the problem at hand. In this work we explore techniques that utilize data generated by a high-fidelity simulation of a dynamical system in order to create a reduced order model, which captures the overall behaviour of the original system. More specifically, we focus on methods that decompose the data into modes that describe underlying dominant dynamics of the system.

1.1 Background

There exist several methods that identify coherent structures from data. Dynamic mode decomposition (DMD) has since its first appearance in [25] become a popular method to analyse data generated from both linear and non-linear dynamical systems. In recent years, efforts have been made to improve the numerics as well as propose new variations of the method. These include more efficient implementations that are based on the singular value decomposition (SVD) instead of Arnoldi iteration and new frameworks where non-sequential data can be utilized [24, 12].

The DMD extracts modes from a data sequence, called the DMD modes and eigenvalues. The data sequence can be generated by sampling either a linear or non-linear dynamical system. The modes are computed from an eigendecomposition of a matrix that is constructed from the data sequence. Each eigenvalue defines a temporal frequency and growth rate of the spatial dynamical characteristics described by its corresponding mode. If the underlying system is linear, these modes can be used to give an exact representation of the system. The same can however not be said of the modes extracted from data that originates from non-linear dynamics.

Even though the DMD modes may not give an exact representation of a non-linear system, they can still be useful as they approximate underlying dominant dynamical behaviours. This was shown by Rowley *et al.* (2009) [24], where they demonstrated a tight connection between DMD and spectral analysis of the Koopman operator. The Koopman operator is an infinite-dimensional linear operator that evolves observables depicting any dynamical system linearly in time. An observable can be seen as a new set of states that originate from the states of the system. Since the Koopman operator is linear

we may similarly to DMD decompose it into spatial-temporal modes, now denoted by the Koopman modes and eigenvalues. Instead of describing the dynamics of the states of the underlying system, as the DMD modes do, the Koopman modes describe the evolution of the observables. Rowley *et al.* [24] showed that the DMD and Koopman modes are equal for linear observables and stated further that DMD can be seen as a numerical approximation of Koopman spectral analysis. From this point of view it is reasonable to use DMD to analyse non-linear systems as well.

Using Koopman operator theory one can transform a non-linear dynamical system into a linear system using a set of observables. Unfortunately, this does not hold for all sets of observables. We can however guarantee that the resulting system is linear by requiring them to span a function space invariant to the Koopman operator. Finding these is far from trivial regardless of if the dynamics are known or unknown. The observables are namely problem-dependant and must be manually prepared based on the underlying dynamics. As a result of increased computational power and an abundance of data, machine learning techniques have become popular and have successfully been applied to many fields of engineering such as image analysis and natural language processing. In this setting, Takeishi *et al.* (2017) [26] proposed a fully data-driven method for learning observables that span a Koopman invariant subspace. This would allow us to transform a non-linear system into a linear system without knowing anything about the underlying dynamics.

Due to both the DMD and Koopman method being equation-free and purely data-driven, they have been widely adopted in many scientific and engineering fields, e.g. fluid mechanics [24] and disease modelling [23]. With equation-free we mean that the methods do not require any a priori knowledge of the underlying differential equations. Characterizing the dynamics of a system by extracting spatial-temporal modes from high-fidelity data is not computationally cheap. However, if the characteristics of the system do not vary in time these modes and eigenvalues only have to be computed once. In this case the methods can be seen as having an offline learning phase where the modes are extracted. Using these modes either an exact or approximated model of the original system can be constructed. This model can in many cases be significantly less computationally expensive to use for online evaluation compared to solving complex differential equations. Here online evaluation refers to the evolution of the states of the system forward in time. Note that computational time and power are not as limited in the offline setting as in online evaluation. Thus, these methods show promise in being integrated into real-time systems where the computing time is of paramount importance.

This work focuses on understanding the DMD and Koopman method and to study their limitations. We employ them on both linear and non-linear systems and study how they can be utilized to create reduced order models from high-fidelity data. We demonstrate the performance of the methods by comparing their reconstructed solutions to high-fidelity simulations. In this context, simulation implies performing numerical time integration of mathematical models describing a dynamical system.

1.2 Problem Formulation

Haldex employs the open-source standardized modelling language Modelica [5] for modelling of physical dynamical systems. These models can quickly become very complex and may have higher fidelity than strictly required for a specific task. This results in them being computationally heavy to simulate and infeasible to run in real-time. The possibility to perform an overall model fidelity reduction is therefore of interest.

The aim of this thesis is thus to investigate if the DMD or Koopman method may be used as a model reduction framework. These methods will be applied to a highly non-linear brake system developed by Haldex in order to create a reduced order model. This model should have sufficient fidelity and require less computing time than the original high-fidelity model.

We now give a mathematical description of the problem at hand. Consider a linear or non-linear dynamical system described by the map $\tilde{\mathbf{f}}$ that evolves on a manifold $\mathcal{M} \subset \mathbb{R}^{n_x}$, $\tilde{\mathbf{f}} : \mathcal{M} \times \mathcal{N} \times \mathbb{R}^+ \to \mathcal{M}$, where $\mathcal{N} \subset \mathbb{R}^{n_u}$. The map $\tilde{\mathbf{f}}$ may be time-dependent. Let $\mathbf{x}(t) \in \mathcal{M}$ denote the states of the system at time t and let $\mathbf{x}_0$ be the initial value of the states at the initial time t_0. Furthermore, let $\mathbf{u}(t) \in \mathcal{N}$ be an input signal to the system with initial value $\mathbf{u}_0$ and let $\theta \in \mathbb{R}^s$ denote the system parameters. The dynamical system can thus be described by the following initial value problem (IVP)

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \tilde{\mathbf{f}}(\mathbf{x}(t), \mathbf{u}(t), t; \theta) \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \\ \mathbf{u}(t_0) &= \mathbf{u}_0. \end{aligned} \tag{1.2.1}$$

Since this thesis revolves around methods that require data sequences, which are sampled from a dynamical system, a discrete formulation of Eq. (1.2.1) is of interest. Consider the discrete linear or non-linear map $\mathbf{f}$ corresponding to $\tilde{\mathbf{f}}$ in Eq. (1.2.1), where $\mathbf{f} : \mathcal{M} \times \mathcal{N} \times \mathbb{Z}^+ \to \mathcal{M}$. Introduce $t = k\Delta t$, where $k \in \mathbb{Z}^+$ and $\Delta t \in \mathbb{R}^+$ denote the time index and step size respectively. Let $\mathbf{x}_k$ denote the states of the system at time index k and $\mathbf{u}_k$ the input signal at the same time index. Then,

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, k; \theta). \tag{1.2.2}$$

The goal of this thesis is to find a reduced order model of $\mathbf{f}$, that is capable of evolving the system forward in time, given only the initial values $\mathbf{x}_0$, $\mathbf{u}_0$ and the input $\mathbf{u}_k$ at each new time step k. This is done in two stages, one offline learning stage and one online evaluation stage. In the offline stage the dynamical system is simulated for different initial values and the generated data is used to create the reduced order model. The model is then used in the online evaluation phase to evolve new unseen initial values forward in time. This will be referred to as predicting the dynamical system.

1.3 Overview

This thesis is organized as follows: § 2 describes the dynamical systems that will be used throughout this thesis. § 3 describes the simulation tools used to simulate these systems. § 4 and § 5 give a mathematical description of the DMD and Koopman method respectively. They also present several examples highlighting their strengths and weaknesses. § 6 explains how neural networks can be utilized in the spectral analysis of the Koopman operator. § 7 presents the data of the brake system used to train and test the models. § 8 presents the results from the DMD and Koopman method as well as a comparison with high-fidelity simulations. § 9 discusses the results. Possible improvements and continuations are also proposed. Finally § 10 presents the conclusions made based on the results. In addition, there is an appendix containing a few results that were not presented in § 8.

Both authors have contributed equally to the theory of the DMD and Koopman method. Once the theory was established, we started working in parallel with the implementations of the methods. David Krantz was mainly responsible for the DMD method and Olof Månsson for the Koopman method.

Chapter 2

Description of Dynamical Systems

Throughout this thesis two problems will be considered. First, a linear dynamical system based on a mechanical scheme. The main advantages of this system is that it can be solved analytically, has an intuitive interpretation and is easy to simulate. Second, a part of Haldex's brake systems, namely the Fast Actuating Brake Valve (FABV), a system described by multiple non-linear physics.

2.1 Damped Dual Mass System

A simple damped dual mass system will now be presented. It consists of two point masses, m_1 and m_2, two springs with spring constants k_1 and k_2 as well as a damper between the two masses with damping coefficient c. In addition to that, an external force $F(t)$ acts on the mass m_2. A visual representation of this system is presented in Fig. 2.1.

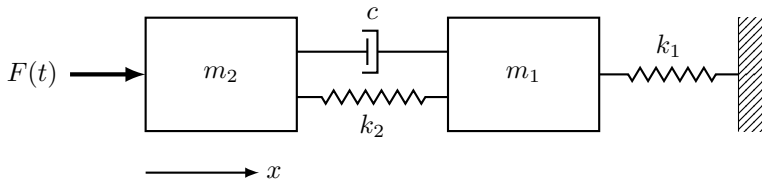


Figure 2.1: The damped dual mass system.

Let x_1 represent the position of the right-hand mass and x_2 the position of the left-hand mass. The state vector $\mathbf{x}$ can for instance be defined as $\mathbf{x} = \begin{bmatrix} x_1 & \dot{x}_1 & x_2 & \dot{x}_2 \end{bmatrix}^T$ (note that one could include more states, such as the accelerations, but it turns out that the accelerations are unnecessary states and will therefore be left out). The dimensions received from this example are $n_x = 4$ and $n_u = 1$, since the input $\mathbf{u}(t) = u(t) = F(t)$, which implies $\mathcal{M} \subset \mathbb{R}^4$ and $\mathcal{N} \subset \mathbb{R}$.

To find an equation similar to Eq. 1.2.2 that describes the dynamics of the system, the equations of motion will be used. For this system they are

$$\begin{aligned} m_1\ddot{x}_1 - c(\dot{x}_2 - \dot{x}_1) - k_2x_2 + (k_1 + k_2)x_1 &= 0 \\ m_2\ddot{x}_2 + c(\dot{x}_2 - \dot{x}_1) + k_2(x_2 - x_1) &= F(t). \end{aligned} \tag{2.1.1}$$

Eq. (2.1.1) can be written in matrix form as

$$\begin{bmatrix} \dot{x}_1 \\ \ddot{x}_1 \\ \dot{x}_2 \\ \ddot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{k_1+k_2}{m_1} & -\frac{c}{m_1} & \frac{k_2}{m_1} & \frac{c}{m_1} \\ 0 & 0 & 0 & 1 \\ \frac{k_2}{m_2} & \frac{c}{m_2} & -\frac{k_2}{m_2} & -\frac{c}{m_2} \end{bmatrix} \begin{bmatrix} x_1 \\ \dot{x}_1 \\ x_2 \\ \dot{x}_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{F(t)}{m_2} \end{bmatrix}, \tag{2.1.2}$$

or more compact as

$$\dot{\mathbf{x}} = A\mathbf{x} + \mathbf{F}(t). \tag{2.1.3}$$

The general solution to discretized versions of equations on the form of Eq. (2.1.3) is given by

$$\mathbf{x}_{k+1} = e^{A\Delta t}\mathbf{x}_k + \int_0^{\Delta t} e^{A\tau}\mathbf{F}(t + \Delta t - \tau)d\tau, \tag{2.1.4}$$

where k denotes time index such that $t = k\Delta t$ and e^A is the exponential of the matrix A. If we assume that $\mathbf{F}(t) = \mathbf{F}_k$ is constant in-between time steps it follows that

$$\mathbf{x}_{k+1} = e^{A\Delta t}\mathbf{x}_k + \left(\int_0^{\Delta t} e^{A\tau} d\tau \right) \mathbf{F}_k. \tag{2.1.5}$$

Since A is invertible this can be further simplified to

$$\mathbf{x}_{k+1} = e^{A\Delta t}\mathbf{x}_k + A^{-1}\left(e^{A\Delta t} - I\right)\mathbf{F}_k, \tag{2.1.6}$$

where I is the identity matrix. In the case of the damped dual mass system, the linear map $\mathbf{f}$ is given by

$$\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) = e^{A\Delta t}\mathbf{x}_k + A^{-1}\left(e^{A\Delta t} - I\right)\mathbf{u}_k, \tag{2.1.7}$$

where $\mathbf{u}_k = \begin{bmatrix} 0 & 0 & 0 & u_k \end{bmatrix}^T$.

2.1.1 Eigenfrequencies

To further understand the damped dual mass system, a method for computing its eigenfrequencies is now presented. First Eq. (2.1.1) is rewritten as

$$M\ddot{\mathbf{x}} + C\dot{\mathbf{x}} + K\mathbf{x} = \mathbf{F}, \tag{2.1.8}$$

where $M, C, K \in \mathbb{R}^{2\times 2}$, $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$ and $\mathbf{F} = \begin{bmatrix} 0 & F(t) \end{bmatrix}^T$. The matrices M, C, K can be identified as

$$M = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}, \quad C = \begin{bmatrix} c & -c \\ -c & c \end{bmatrix}, \quad K = \begin{bmatrix} k_1 + k2 & -k_2 \\ -k_2 & k_2 \end{bmatrix}. \tag{2.1.9}$$

We make the ansatz that the displacement of the two masses can be described by a harmonic function, namely $\mathbf{x} = \tilde{\mathbf{x}}e^{i\omega t}$, where $\tilde{\mathbf{x}}$ denotes an amplitude and ω describes the frequency and the dampening of the oscillation. Furthermore, since the eigenfrequencies of the system do not depend on the input, set $\mathbf{F} = \mathbf{0}$ [9]. Inserting the ansatz into Eq. (2.1.8) gives

$$(-\omega^2 M + i\omega C + K)\mathbf{x} = \mathbf{0}. \tag{2.1.10}$$

For this equation to have a non-trivial solution it is required that

$$\det\left(-\omega^2 M + i\omega C + K\right) = 0, \tag{2.1.11}$$

or equivalently

$$m_1m_2\omega^4 - ic(m_1+m_2)\omega^3 + ((-k_1-k_2)m_2 - m_1k_2)\omega^2 + ick_1\omega + k_1k_2 = 0. \tag{2.1.12}$$

This gives two pairs of complex conjugated solutions ω_j, where $j = 1, 2$. Dividing ω_j into its real and imaginary part as $\omega_j = a_j \pm b_j i$, where $a_j, b_j \in \mathbb{R}$, and inserting them into the ansatz yields

$$\mathbf{x} = \tilde{\mathbf{x}}e^{i\omega_j t} = \tilde{\mathbf{x}}e^{i(a_j \pm b_j i)t} = \tilde{\mathbf{x}}e^{\mp b_j t} \cdot e^{ia_j}. \tag{2.1.13}$$

The interpretation of this is that the masses oscillate with the angular frequency a_j and experience exponential dampening described by b_j. In summary, the eigenfrequencies of the damped dual mass system, f_j, are found as

$$f_j = \frac{\mathrm{Re}(\omega_j)}{2\pi}. \tag{2.1.14}$$

2.1.2 Non-linear Damped Dual Mass System

As an intermediate step between the system in Fig. 2.1 and the more complex system in the next section, a non-linear version of the damped dual mass system is considered. To make the damped dual mass system non-linear, friction $F_\mu(\dot{x}_2)$ is introduced for the second mass as if it is sliding on a rough surface. This results in non-linear behaviour. The equations of motion now become

$$\begin{aligned} &m_1\ddot{x}_1 - c(\dot{x}_2 - \dot{x}_1) - k_2x_2 + (k_1+k_2)x_1 = 0 \\ &m_2\ddot{x}_2 + c(\dot{x}_2 - \dot{x}_1) + F_\mu(\dot{x}_2) + k_2(x_2 - x_1) = F(t). \end{aligned} \tag{2.1.15}$$

2.2 Fast Actuating Brake Valve

The Fast Actuating Brake Valve (FABV) is one part of a brake system developed by Haldex. It consists of several electrical, magnetic and mechanical components, and is highly non-linear. The FABV takes two set point currents as inputs, $I_1(t)$, $I_2(t) \in \mathbb{R}$, and the output is the applied brake torque. In this thesis the torque will not be used as the output, but rather the pressure p that yields the torque. A schematic sketch of the considered FABV model is given in Fig. 2.2. $I_1(t) \geq 0$ and $I_2(t) \leq 0$ are discontinuous functions and are not simultaneously nonzero. Their effect can be thought of as the increase and decrease of the rate of change of the brake torque. The input signals are fed through two similar subsystems, each with four subsystems of their own. The set point current together with the system voltage $U_{sys} \in \mathbb{R}$ is input to a DC-DC converter. The output current from the converter is then sent to a magnetic circuit that is connected to a mechanical system. These two subsystems give rise to a force that together with either the system or atmospheric pressure acts on a valve. The output from the valve is a flow that gives rise to a pressure, $p \in \mathbb{R}$, that we regard as the output of the FABV system.

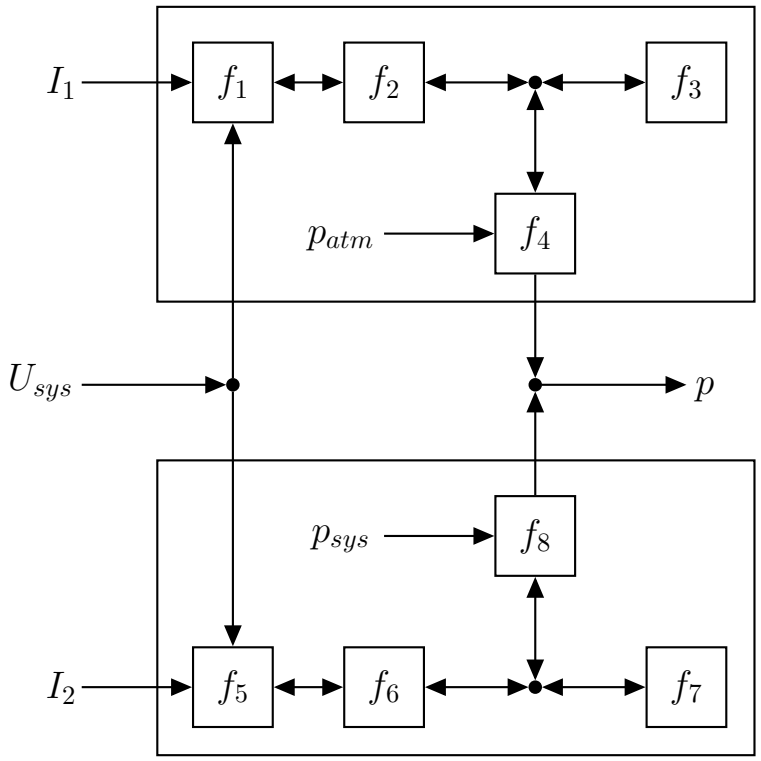


Figure 2.2: *Sketch of the FABV system with inputs I_1 and I_2, and output p. The system is divided into eight subsystems where f_1, f_5 are the DC-DC converters, f_2, f_6 the magnetic circuits, f_3, f_7 the mechanical systems and f_4, f_8 are the valve components. U_{sys} denotes the system voltage, p_{atm} the atmospheric pressure and p_{sys} the system pressure.*

The number of internal states, $\mathbf{x}$, of the FABV model is $n_x \approx 2400$ and since it has two inputs we get $n_u = 2$. Consider the internal states as $\mathbf{x} = \begin{bmatrix}\mathbf{x}_1^T & \mathbf{x}_2^T & \cdots & \mathbf{x}_8^T\end{bmatrix}^T$ where $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_8$ are states of the eight different subsystems in Fig. 2.2, each with their own map $\mathbf{f}_1$, $\mathbf{f}_2$, ..., $\mathbf{f}_8$. The system can then be described by the following systems of differential algebraic equations (DAEs)

$$\begin{cases} \dot{\mathbf{x}}_1 = \mathbf{f}_1(\mathbf{x}_1, \mathbf{x}_2, I_1, t; U_{sys}) \\ \dot{\mathbf{x}}_2 = \mathbf{f}_2(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, t) \\ \dot{\mathbf{x}}_3 = \mathbf{f}_3(\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, t) \\ \dot{\mathbf{x}}_4 = \mathbf{f}_4(\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, t; p_{atm}) \\ \mathbf{0} = \mathbf{g}_1(\mathbf{x}_1, \mathbf{x}_2) \\ \mathbf{0} = \mathbf{g}_2(\mathbf{x}_2) \end{cases} \qquad \begin{cases} \dot{\mathbf{x}}_5 = \mathbf{f}_5(\mathbf{x}_5, \mathbf{x}_6, I_2, t; U_{sys}) \\ \dot{\mathbf{x}}_6 = \mathbf{f}_6(\mathbf{x}_5, \mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8, t) \\ \dot{\mathbf{x}}_7 = \mathbf{f}_7(\mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8, t) \\ \dot{\mathbf{x}}_8 = \mathbf{f}_8(\mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8, t; p_{sys}) \\ \mathbf{0} = \mathbf{g}_1(\mathbf{x}_5, \mathbf{x}_6) \\ \mathbf{0} = \mathbf{g}_2(\mathbf{x}_6), \end{cases} \tag{2.2.1}$$

where $\mathbf{g}_1$ describes Kirchhoff's circuit laws, and $\mathbf{g}_2$ describes Ampère's circuital law and Faraday's law of induction. This gives the manifold $\mathcal{M} = \{\mathbf{x} \in \mathbb{R}^{n_x} : \mathbf{0} = \mathbf{g}_1(\mathbf{x}_1, \mathbf{x}_2), \mathbf{0} = \mathbf{g}_2(\mathbf{x}_2), \mathbf{0} = \mathbf{g}_1(\mathbf{x}_5, \mathbf{x}_6), \mathbf{0} = \mathbf{g}_2(\mathbf{x}_6)\}$.

Since this thesis revolves around data-driven methods, a more detailed description of the FABV is not needed. This section mainly served the purpose of providing a general understanding of the FABV model.

Chapter 3

Simulation Tools

In this chapter we give a brief introduction to the software based simulation tools utilized in this thesis, namely Functional Mock-up Interface (FMI), Functional Mock-up Unit (FMU) and co-simulation.

3.1 Functional Mock-Up Interface Overview

Simulation tools can have different ways of representing a mathematical model of a dynamical system. Problems arise when one tries to import a model from one simulation tool to another and simulate it there. The FMI provides a standardized interface for system modelling and was developed with the purpose of solving these compatibility issues. The latest iteration of the FMI is the 2.0.1 standard [4]. The two main features of this standard are: FMI for model exchange and FMI for co-simulation. Models compatible with the FMI standard are called FMUs, and can be of the type model exchange or co-simulation. The main difference between the two is that when simulating a model exchange FMU, an external solver must be supplied. A co-simulation FMU however, contains its own solver and thus no external solver is needed to simulate it. FMI for co-simulation also allows for coupling of two or more FMUs with solvers in a joint environment. The models are solved independently from each other using their own built-in solver and they exchange data using input and output connections [7]. This opens up the possibility to employ specialized solvers for different subsystems, which may yield an increased simulation performance [3]. It is possible to co-simulate two or more FMUs using just the external solver.

The FMI standard is currently maintained by the Modelica Association, which in addition to the standard, provides open-source software for modelling. More specifically, the object-oriented, equation-based programming language Modelica and the Modelica Standard Library containing a wide range of standard components that can be used to model systems in several engineering domains, e.g. mechanical, electrical and thermal. These components include e.g. models of springs, resistors and valves but also mathematical operators such as addition and the absolute value. Thus, the Modelica language can be used to model dynamical systems such as the ones described in Ch. 2. One way to do this is to utilize a Modelica simulation environment. In this thesis we use Dymola and OpenModelica. These programs have graphical interfaces where one can build Modelica models by dragging and dropping Modelica components into an environment and make connections between them. They can also be used to simulate Modelica models and FMUs using solvers such as DASSL and CVode. The Modelica models and FMUs are compiled into C code when simulated. Another feature of these programs that was frequently used throughout this thesis was exporting a Modelica model into an FMU. This task

was exclusively done in Dymola, since OpenModelica does not support exporting co-simulation FMUs compatible with the FMI standard 2.0.

The simulations performed in this thesis were not done in either of the two mentioned simulation environments. Instead, the Python package PyFMI [3] was used for all simulations due to the possibility to easily make modifications and experiments using code. The package supports handling and simulating both model exchange and co-simulation FMUs compatible with the FMI standard 2.0. For example, using PyFMI we can change the value of a parameter in a certain component of the FMU and then directly simulate the modified FMU, without having to leave Python. When simulating a model exchange FMU, PyFMI connects to the Python package Assimulo [2], which provides PyFMI with the solver needed to perform the simulation. An illustration of the mentioned software utilities is presented in Fig. 3.1.

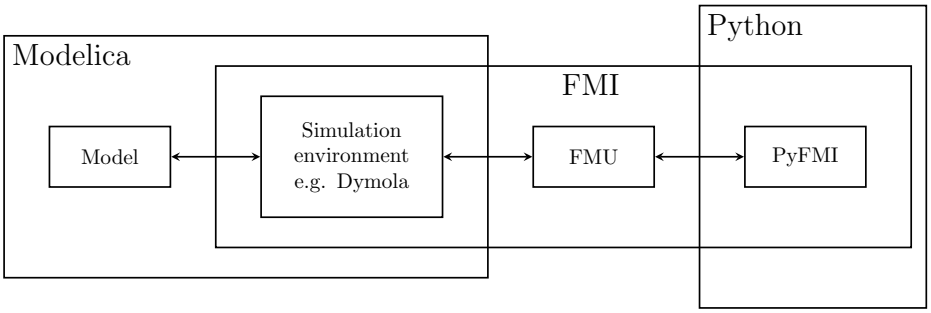


Figure 3.1: *An illustration of how the different software utilities used for this thesis interact with each other.*

The workflow used to simulate the FABV is described in further detail in Ch. 7.

Chapter 4

Dynamic Mode Decomposition

Dynamic mode decomposition (DMD) is a numerical method used to analyse linear and non-linear dynamical systems. DMD utilizes the concept of modal decomposition to extract dynamical characteristics of a system. These characteristics are extracted from simulation data. This enables online evaluation of the system, but the learnt characteristics can aid in other types of analysis as well.

4.1 Dynamic Mode Decomposition

The dynamical systems studied in this thesis can in their most general form be modelled as differential algebraic equations (DAEs) on the form of Eq. (1.2.1). The data generated from simulations of these models can be thought of as being generated from the discrete system described in (1.2.2). We introduce the data pairs $(\mathbf{x_k}, \mathbf{y_k})$, which are related as

$$\mathbf{y}_k = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, k), \tag{4.1.1}$$

where θ has been excluded compared to Eq. (1.2.2). To simplify the following analysis, $\mathbf{f}$ is assumed to be an autonomous time-independent linear map represented by a matrix $A \in \mathbb{R}^{n_x \times n_x}$. Eq. (4.1.1) then becomes

$$\mathbf{y}_k = A\mathbf{x}_k. \tag{4.1.2}$$

In many practical applications the underlying system matrix A is unknown and the only thing we are able to measure is different input and output data pairs $(\mathbf{x}_k, \mathbf{y}_k)$. As proposed by Tu *et al.* (2014) [12], we now introduce DMD as a way of analysing these data pairs. This definition uses the Moore-Penrose pseudoinverse of a matrix as defined in Def. 4.1.2 [15]. We furthermore denote the complex conjugate transpose of a matrix using a superscript H.

Definition 4.1.1 (Dynamic Mode Decomposition)**.** Given m data pairs $(\mathbf{x}_k, \mathbf{y}_k)$, also called snapshots, construct $X \in \mathbb{R}^{n_x \times m}$ and $Y \in \mathbb{R}^{n_x \times m}$ according to

$$X = \begin{bmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_m \end{bmatrix}, \quad Y = \begin{bmatrix} \mathbf{y}_1 & \cdots & \mathbf{y}_m \end{bmatrix}. \tag{4.1.3}$$

Define the matrix $\hat{A} \in \mathbb{R}^{n_x \times n_x}$ as

$$\hat{A} = YX^{\dagger}, \tag{4.1.4}$$

where $X^{\dagger} \in \mathbb{R}^{m \times n_x}$ denotes the Moore-Penrose pseudoinverse of X as defined by Def. 4.1.2. The *dynamic mode decomposition* of the pair (X, Y) is given by the eigendecomposition of $\hat{A}$. That is, the eigenvectors and eigenvalues of $\hat{A}$ are the DMD modes and eigenvalues.

Definition 4.1.2 (Moore-Penrose Pseudoinverse)**.** Consider a matrix $X \in \mathbb{R}^{n_x \times m}$, where $n_x \leq m$, and its reduced singular value decomposition (SVD) $X = U\Sigma V^H$, where $U \in \mathbb{R}^{n_x \times n_x}$, $\Sigma \in \mathbb{R}^{n_x \times n_x}$ and $V \in \mathbb{R}^{m \times n_x}$. The *Moore-Penrose pseudoinverse* of X, denoted by $X^\dagger$, is then defined as

$$X^\dagger = V\Sigma^{-1}U^H. \tag{4.1.5}$$

Remark 4.1.1. This formulation of the DMD allows for non-sequential data, meaning that the data pairs $(\mathbf{x}_k, \mathbf{y}_k)$ can come in any order and be sampled non-uniformly in time. In other words, the columns of X has to be in no particular order as long as the columns of Y are in the exact same order.

Remark 4.1.2. The interpretation of Eq. (4.1.4) is that if X has full rank, then $\hat{A}X = Y$ holds exactly and $\hat{A}$ is found by minimizing $\|\hat{A}X - Y\|_F$ as shown below, where $\|\cdot\|_F$ denotes the Frobenius norm. In the case when the data pairs come from a linear dynamical system as in Eq. (4.1.2), $\hat{A}$ defined via Eq. (4.1.4) coincides with the system matrix A. Worth mentioning is that if X does not have full rank the minimization problem does not have a unique solution. To get a unique solution one can impose an additional condition, e.g. that $\hat{A}$ should minimize $\|\hat{A}\|_F$. We now show that $\hat{A}$ is found by solving the following minimization problem

$$\begin{aligned}\hat{B} = \underset{B \in \mathbb{R}^{n_x \times n_x}}{\arg\min} \|BX - Y\|_F &= \underset{B \in \mathbb{R}^{n_x \times n_x}}{\arg\min} \|BX - Y\|_F^2 \\ &= \underset{B \in \mathbb{R}^{n_x \times n_x}}{\arg\min} \operatorname{Tr}\left((BX - Y)^H (BX - Y)\right).\end{aligned} \tag{4.1.6}$$

The solution is found by differentiating with respect to B, where we use that

$$\frac{\partial \mathrm{Tr}(F(B))}{\partial B} = f(B)^H, \tag{4.1.7}$$

where $f(\cdot)$ is the scalar derivative of $F(\cdot)$ [22]. Setting the received expression equal to the zero matrix the gives

$$0 = \left(2BXX^H - 2YX^H\right)^H \Rightarrow \hat{B} = YX^H\left(XX^H\right)^{-1} = YX^\dagger, \tag{4.1.8}$$

where the right pseudoinverse of X was identified. Lastly, we note that $\hat{B} = \hat{A}$.

If $\hat{A}$ given by (4.1.4) has a full set of right eigenvectors $\mathbf{v}_i \in \mathbb{C}^{n_x}$ with corresponding eigenvalues $\lambda_i \in \mathbb{C}$ it holds that

$$\mathbf{x}_k = \sum_{i=1}^{n_x} \langle \mathbf{w}_i, \mathbf{x}_k \rangle \mathbf{v}_i, \tag{4.1.9}$$

where $\mathbf{w}_i \in \mathbb{C}^{n_x}$ denotes the left eigenvectors of $\hat{A}$, i.e. $\mathbf{w}_i^H \hat{A} = \lambda_i \mathbf{w}_i^H$ [24]. Applying $\hat{A}$ from the left to Eq. (4.1.9) gives

$$\hat{\mathbf{y}}_k \coloneqq \hat{A}\mathbf{x}_k = \sum_{i=1}^{n_x} \lambda_i \langle \mathbf{w}_i, \mathbf{x}_k \rangle \mathbf{v}_i. \tag{4.1.10}$$

This allows us to compute the states in the next time step $k+1$, given the states at time k.

Remark 4.1.3. We want to emphasize that, as seen in Eq. (4.1.10), there are two different ways to evolve the states in time, which are mathematically equivalent. One may either directly use the matrix $\hat{A}$ or use the DMD modes and eigenvalues. We will later see how one can compute the DMD modes and eigenvalues in a more efficient way, both from a computational and a memory perspective.

In addition to being able to predict future states, the DMD eigenvalues can be used to analyse the dynamics of a system. Each eigenvalue contains information about the exponential growth rate and

oscillation of its corresponding mode. A damped oscillation can be described as $e^{-\mu t+2\pi i f t}$, where f is the frequency of the oscillation and μ is the damping coefficient for that frequency. In the discrete case this results in $e^{-\mu k\Delta t+2\pi i f k\Delta t} = (e^{-\mu\Delta t+2\pi i f\Delta t})^k$, where we have used that $t = k\Delta t$. The DMD eigenvalues can therefore be expressed as

$$\lambda_j = e^{-\mu_j\Delta t+2\pi i f_j\Delta t}, \tag{4.1.11}$$

which implies that λ_j^k describes the oscillation and dampening at time $k\Delta t$. The frequency of the j:th DMD mode can thus be found by solving Eq. (4.1.11) for the frequency f_j, which gives

$$f_j = \frac{\mathrm{Im}(\log(\lambda_j))}{2\pi\Delta t}. \tag{4.1.12}$$

We now study under which circumstances $\hat{A}$ satisfies $Y = \hat{A}X$ by introducing the following definition and theorem proposed by Tu *et al.* [12]. We will see that the concept of linear consistency, which is a weaker condition on the data for $Y = \hat{A}X$ to hold than discussed in Rem. 4.1.2, allows us to find an exact representation of the linear system as $\hat{A}$ defined by Eq. (4.1.4). The importance of having linearly consistent data is highlighted further in Ex. 4.3.2.

Definition 4.1.3 (Linear consistency)**.** The matrix $X \in \mathbb{R}^{n_x\times m}$ is *linearly consistent* with the matrix $Y \in \mathbb{R}^{n_x\times m}$ if $\mathscr{N}(X) \subset \mathscr{N}(Y)$, where $\mathscr{N}(X)$ denotes the null-space of X.

Theorem 4.1.1. *Let $\hat{A}$ be defined by Eq.* (4.1.4)*. Then, $Y = \hat{A}X$ if and only if X is linearly consistent with Y.*

Proof. Suppose that $\hat{A}X = Y$ and let $\mathbf{z} \in \mathbb{R}^{n_x}$ belong to the null-space of X. It follows that

$$X\mathbf{z} = \mathbf{0} \Rightarrow \hat{A}X\mathbf{z} = \mathbf{0} \Leftrightarrow Y\mathbf{z} = \mathbf{0}, \tag{4.1.13}$$

which means that $\mathscr{N}(X) \subset \mathscr{N}(Y)$. This is precisely the definition of linear consistency between X and Y.

Now, assume that X is linearly consistent with Y. That is, for all vectors $\mathbf{z} \in \mathscr{N}(X)$ it holds that

$$\begin{cases} X\mathbf{z} = \mathbf{0} \\ Y\mathbf{z} = \mathbf{0} \end{cases} \Rightarrow \begin{cases} \hat{A}X\mathbf{z} = \mathbf{0} \\ Y\mathbf{z} = \mathbf{0}. \end{cases} . \tag{4.1.14}$$

Subtracting the bottom row from the top row in Eq. 4.1.14 yields

$$(Y - \hat{A}X)\mathbf{z} = \mathbf{0}. \tag{4.1.15}$$

Consider the expression inside the parenthesis

$$Y - \hat{A}X = Y - YX^\dagger X = Y(I - X^\dagger X), \tag{4.1.16}$$

where $I \in \mathbb{R}^{m\times m}$ denotes the identity matrix. $X^\dagger X$ is the orthogonal projection onto the range space of X^H, which implies that $I - X^\dagger X$ is a projection onto a space orthogonal to the range space of X^H, i.e. $\mathscr{N}(X)$. Since $\mathscr{N}(X) \subset \mathscr{N}(Y)$ this gives $Y(I - X^\dagger X) = 0$, and therefore

$$Y - \hat{A}X = 0 \Leftrightarrow Y = \hat{A}X. \tag{4.1.17}$$

□

Remark 4.1.4. If X is not linearly consistent with Y then $\hat{A}X - Y \neq 0$. A potential solution to this issue suggested by Tu *et al.* [12] is to append time-shifted data to the data matrix $X \in \mathbb{R}^{n_x q \times (m-q)}$, also called time delay embedding, as

$$X = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_{m-q-1} \\ \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_{m-q} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_q & \mathbf{x}_{q+1} & \cdots & \mathbf{x}_{m-1} \end{bmatrix} \tag{4.1.18}$$

and analogously for $Y \in \mathbb{R}^{n_x q \times (m-q)}$, see Ex. 4.3.4. It is however not obvious how many data pairs are needed to be appended for the matrices to become linearly consistent. Time delay embeddings can also be used to improve robustness of DMD in cases where Eq. (4.1.2) only holds approximately, for instance where there is random noise added to the data. Worth mentioning is that even if X is not linearly consistent with Y the DMD modes and eigenvalues might still provide useful information, as we will see in Ex. 4.3.2.

We now show an example of how the DMD is able to find the underlying system matrix from data generated by a linear system.

Example 4.1.1 (DMD of data from a linear system)**.** Consider the damped dual mass system in Sect. 2.1 without any input and construct the data matrices X, Y according to Eq. (4.1.3) using the analytical expression in Eq. (2.1.6) to generate the snapshots. Due to the problem being linear, these data matrices will be linearly consistent. By Thm. 4.1.1, we then expect $\hat{A}$ defined by Eq. (4.1.4) to coincide with $e^{A\Delta t}$ in Eq. (2.1.4). Computing $\hat{A}$ and comparing it with A gives that $\|\hat{A} - A\|_F \approx 10^{-14}$, meaning that we get the expected result. Note that this result holds for any values of parameters of the system. Worth mentioning is that the sampling frequency, i.e. $1/\Delta t$, has to be high enough in order for $\hat{A}$ to be able to reconstruct the highest frequency of the system. The number of snapshots also have to be $m > n_x = 4$ since there are four states.

We now present an algorithm describing how to perform DMD and use it to predict future states of some dynamical system.

Algorithm 1.

1. Construct X and Y according to Eq. (4.1.3) from the data pairs $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$.
2. Compute the (reduced) SVD of X, i.e. $X = U\Sigma V^H$.
3. Set $\hat{A} = YX^\dagger = YV\Sigma^{-1}U^H$.
4. Compute the eigenvalues and the left and right eigenvectors of $\hat{A}$, such that $\hat{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$ and $\mathbf{w}_i^H \hat{A} = \lambda_i \mathbf{w}_i^H$ for $i = 1, \ldots, n_x$.
5. Scale the eigenvectors such that $\langle \mathbf{w}_i, \mathbf{v}_j \rangle = \delta_{ij}$, where δ is the Kronecker delta.
6. Perform prediction given $\mathbf{x}_k$ according to

$$\hat{\mathbf{y}}_k = \sum_{i=1}^{n_x} \lambda_i \langle \mathbf{w}_i, \mathbf{x}_k \rangle \mathbf{v}_i$$

We want to point out that if you are only interested in evolving the states of the system in time, then this algorithm is not computationally efficient. If you are able to directly use $\hat{A}$, then, as discussed in

Rem. 4.1.3, you do not need to compute the DMD modes and eigenvalues. However, if it is not feasible to explicitly compute and store $\hat{A}$, then the remedy is to use Alg. 2 instead. With this algorithm one can find the DMD modes and eigenvalues without explicitly having to compute $\hat{A}$. This is achieved by utilizing projection and a rank r truncation of X, where r is the rank of X as described by the following theorem. The DMD modes computed by Alg. 2 are simply the DMD modes from Alg. 2 projected onto the range space of X, and will be referred to as the projected DMD modes. The theorem is based on the work done by Tu *et al.* [12].

Theorem 4.1.2. *Consider two data matrices X, Y and $\hat{A}$ defined as in Def. 4.1.1. Let X be of rank r. Then, the SVD of X truncated to rank r is $X = U_r \Sigma_r V_r^H$, with $U_r \in \mathbb{R}^{n_x \times r}$, $\Sigma_r \in \mathbb{R}^{r \times r}$, $V_r \in \mathbb{R}^{m \times r}$. Furthermore, let $P = U_r U_r^H$ denote the orthogonal projection onto the range space of X and the matrix $\tilde{A} = U_r^H \hat{A} U_r \in \mathbb{R}^{r \times r}$. If $\tilde{\mathbf{v}}_i \in \mathbb{C}^r$ and $\tilde{\mathbf{w}}_i \in \mathbb{C}^r$ for $i = 1, 2, \ldots, r$ are the left and right eigenvectors of $\tilde{A}$ with corresponding eigenvalue $\tilde{\lambda}_i \in \mathbb{C}$, then the left and right eigenvectors of $P\hat{A} \in \mathbb{R}^{n_x \times n_x}$ are $\mathbf{v}_i = U_r \tilde{\mathbf{v}}_i \in \mathbb{C}^{n_x}$ and $\mathbf{w}_i = U_r \tilde{\mathbf{w}}_i \in \mathbb{C}^{n_x}$, respectively, with eigenvalues $\tilde{\lambda}_i$.*

Proof. For the right eigenvectors we have that $\tilde{A}\tilde{\mathbf{v}}_i = \tilde{\lambda}_i \tilde{\mathbf{v}}_i$, for $i = 1, 2, \ldots, r$. We wish to find $\mathbf{v}_i$ and λ_i such that $P\hat{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$. From $\mathbf{v}_i = U_r \tilde{\mathbf{v}}_i$, we get that

$$P\hat{A}\mathbf{v}_i = U_r \underbrace{U_r^H \hat{A} U_r}_{=\tilde{A}} \tilde{\mathbf{v}}_i = U_r \tilde{A} \tilde{\mathbf{v}}_i = \tilde{\lambda}_i U_r \tilde{\mathbf{v}}_i = \lambda_i \mathbf{v}_i. \tag{4.1.19}$$

What is left to show is that $\mathbf{v}_i = U_r \tilde{\mathbf{v}}_i \neq \mathbf{0}$. By expanding U_r into its column vectors $\mathbf{u}_i$ and $\tilde{\mathbf{v}}_i$ into its elements $\tilde{v}_i$, this expression can be rewritten as

$$\mathbf{v}_i = U_r \tilde{\mathbf{v}}_i = \begin{bmatrix} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_r \\ | & | & & | \end{bmatrix} \begin{bmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \vdots \\ \tilde{v}_r \end{bmatrix} = \sum_{i=1}^{r} \tilde{v}_i \mathbf{u}_i. \tag{4.1.20}$$

Since the columns $\mathbf{u}_i$ in U_r are linearly independent, the only way this can be equal to the zero vector is if $\tilde{v}_i = 0$ for $i = 1, 2, \ldots, r$. This would however contradict the fact that $\tilde{\mathbf{v}}_i$ is an eigenvector. Hence, we have shown that $\mathbf{v}_i = U_r \tilde{\mathbf{v}}_i$ is a right eigenvector of $P\hat{A}$ with eigenvalue $\lambda_i = \tilde{\lambda}_i$.

The proof for the left eigenvector is performed analogously. For these we have that $\tilde{\mathbf{w}}_i^H \tilde{A} = \tilde{\lambda}_i \tilde{\mathbf{w}}_i^H$. The goal is to find $\mathbf{w}_i$ and λ_i such that $\mathbf{w}_i^H P\hat{A} = \lambda_i \mathbf{w}_i^H$. Let $\mathbf{w}_i = U_r \tilde{\mathbf{w}}_i$, then

$$\mathbf{w}_i^H P\hat{A} = (U_r \tilde{\mathbf{w}}_i)^H U_r U_r^H \hat{A} = \tilde{\mathbf{w}}_i^H \underbrace{U_r^H U_r}_{=I} \underbrace{U_r^H Y V_r \Sigma_r^{-1}}_{=\tilde{A}} U_r^H = \tilde{\mathbf{w}}_i^H \tilde{A} U_r^H = \tilde{\lambda}_i \tilde{\mathbf{w}}_i^H U_r^H = \lambda_i \mathbf{w}_i. \tag{4.1.21}$$

The proof that $\mathbf{w}_i \neq \mathbf{0}$ can be shown exactly as with the left eigenvector. Conclusively, we have proven that $\mathbf{w}_i = U_r \tilde{\mathbf{w}}_i$ is a left eigenvector of $P\hat{A}$ with eigenvalue $\lambda_i = \tilde{\lambda}_i$. □

If the SVD of X instead was truncated to rank ν where $\nu < r$, then we get an approximation of X. The question of how good this approximation is for a given ν thus emerges. As stated in the following theorem, this precision can be quantified using a singular value of X [28].

Theorem 4.1.3. *Let r denote the rank of $X \in \mathbb{R}^{n_x \times m}$ and σ_i its singular values, where $i = 1, 2, \ldots, \min(n_x, m) = p$. For any ν with $0 \leq \nu \leq r$, define $X_\nu = U_\nu \Sigma_\nu V_\nu^H$ as X truncated to rank ν using its SVD. As a special case, if $\nu = p$, define $\sigma_{\nu+1} = 0$. Then,*

$$\|X - X_\nu\|_2 = \sigma_{\nu+1}. \tag{4.1.22}$$

Proof. Write X and X_ν as sums of rank-one matrices in terms of the singular values σ_i of X with corresponding left and right singular vectors, $\mathbf{u}_i$ and $\mathbf{v}_i$ respectively,

$$X = \sum_{i=1}^{p} \sigma_i \mathbf{u}_i \mathbf{v}_i^H, \quad X_\nu = \sum_{i=1}^{\nu} \sigma_i \mathbf{u}_i \mathbf{v}_i^H. \tag{4.1.23}$$

The norm of the difference between the two matrices now becomes

$$\|X - X_\nu\|_2 = \left\| \sum_{i=1}^{p} \sigma_i \mathbf{u}_i \mathbf{v}_i^H - \sum_{i=1}^{\nu} \sigma_i \mathbf{u}_i \mathbf{v}_i^H \right\|_2 = \left\| \sum_{i=\nu+1}^{p} \sigma_i \mathbf{u}_i \mathbf{v}_i^H \right\|_2 = \sigma_{\nu+1}, \tag{4.1.24}$$

where we have used that for any matrix A, the 2-norm is given by $\|A\|_2 = \sigma_1$, where σ_1 is the largest singular value of A. □

Algorithm 2.

1. Construct X and Y according to Eq. (4.1.3) from the data pairs $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$.
2. Compute the (reduced) SVD of X and truncate it to rank r, i.e. $X = U_r \Sigma_r V_r^H$.
3. Define $\tilde{A} = U_r^H \hat{A} U_r = U_r^H Y X_r^\dagger U_r = U_r^H Y V_r \Sigma_r^{-1}$, $\tilde{A} \in \mathbb{R}^{r \times r}$.
4. Compute the eigenvalues and the left and right eigenvectors of $\tilde{A}$, such that $\tilde{A}\tilde{\mathbf{v}}_i = \lambda_i \tilde{\mathbf{v}}_i$ and $\tilde{\mathbf{w}}_i^H \tilde{A} = \lambda_i \tilde{\mathbf{w}}_i^H$ for $i = 1, \ldots, r$.
5. Project the eigenvectors such that they correspond to the projected DMD modes as $\mathbf{v}_i = U_r \tilde{\mathbf{v}}_i$ and $\mathbf{w}_i = U_r \tilde{\mathbf{w}}_i$ for $i = 1, \ldots, r$.
6. Scale the eigenvectors such that $\langle \mathbf{w}_i, \mathbf{v}_j \rangle = \delta_{ij}$, where δ is the Kronecker delta.
7. Perform prediction given $\mathbf{x}_k$ according to

$$\hat{\mathbf{y}}_k = \sum_{i=1}^{r} \lambda_i \langle \mathbf{w}_i, \mathbf{x}_k \rangle \mathbf{v}_i$$

These algorithms allow for both sequential and non-sequential data. The benefit of using non-sequential data is twofold. Firstly, instead of sampling the solution on a uniform time grid we can choose to sample it on e.g. the internal time steps of the used variable-step solver. Secondly, having non-sequential data also allows one to construct the data matrices with data pairs from different simulation runs performed at different times.

4.2 Handling of Input Signal

Up until this point there has been no explanation for how the DMD method can be applied to a system with an input signal, e.g. the damped dual mass system from Fig. 2.1. One easy solution is to treat the input as one of the states. Simply append the input signal $\mathbf{u}$ to the end of the state vector $\mathbf{x}$. However, one might not be interested in the prediction of the input signal in the next time step. It might not even be possible if the input is a control signal for instance. The solution is to set the corresponding elements in Y to zero. With state vector $\mathbf{x}_k$ and a vector-valued input $\mathbf{u}_k$ we get

$$X = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_m \end{bmatrix}, \quad Y = \begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_m \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}. \tag{4.2.1}$$

If the data matrices X, Y are constructed in this way, the rest of the DMD algorithm can be applied in the same way as before.

4.3 Numerical Examples

In the following examples we consider the damped dual mass system in Fig. 2.1 with initial conditions $\mathbf{x}_0 = \begin{bmatrix} 0 & 0 & 0.1 & -0.2 \end{bmatrix}$ and $k_1 = 250$ N/m, $k_2 = 1000$ N/m, $m_1 = 3$ kg, $m_2 = 2$ kg and $c = \sqrt{1000/2} = 10\sqrt{5}$ kg/s. Inserting these parameters into Eq. (2.1.12) gives the eigenfrequencies of the system as $f_1 = 1.10$ Hz and $f_2 = 4.45$ Hz.

For most of the following examples, snapshots were generated up to time $t = 20$ s. The first 10 s are used to to construct the data matrices X, Y and the remaining 10 s are used for validation. The prediction errors will be calculated over the whole time period.

Example 4.3.1 (DMD of data from a linear system with input)**.** Consider the damped dual mass system in Sect. 2.1 with the data matrices X, Y constructed according to Eq. (4.1.3) using the analytical expression in Eq. (2.1.6) to generate the snapshots. The input $F(t)$ is set to be a sine function with amplitude 10 and frequency 1 Hz. The number of snapshots used for training was $m = 250$. Since this is a linear system we expect the predictions to follow the analytical solution closely.

In Fig. 4.1 the analytical solution, more specifically the position of the second mass, is plotted against the DMD predictions based on analytical data. Not only do they overlap perfectly in the figure, the prediction error $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2 \approx 10^{-12}$, where $\hat{\mathbf{x}}_2$ is the position of the second mass calculated with DMD for all times, and $\mathbf{x}_2^{an}$ is the sampled analytical solution for the second mass.

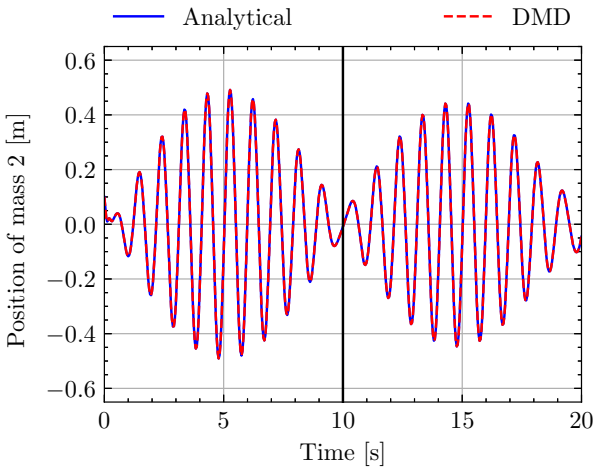


Figure 4.1: *Analytical solution (solid blue) compared to prediction (dashed red) of the position of mass 2 based on analytical data. Data up to time* 10 *s (the solid black line) was used for training.*

Example 4.3.2 (DMD of data that is not linearly consistent)**.** We now study the difference in the precision of the predictions when constructing X, Y from analytical and simulated data. As seen in Ex. 4.1.1, when using analytical data the predictions coincide with the analytical solution. However, when dealing with simulated data the integrator will introduce numerical errors, and thus we do not expect the predictions to be on the same level of accuracy as the ones in Ex. 4.3.1.

We set the same input signal as in Ex. 4.3.1 as well as the same number of snapshots. The system was simulated using the variable-order, variable-step multi-step algorithm CVode. The multi-step method used was a Backward Differentiation Formula (BDF) of maximal order 5. The absolute and relative tolerances were set to 10^{-8} and 10^{-6} respectively.

Following the steps in Alg. 1 we get the predictions shown in Fig. 4.2.

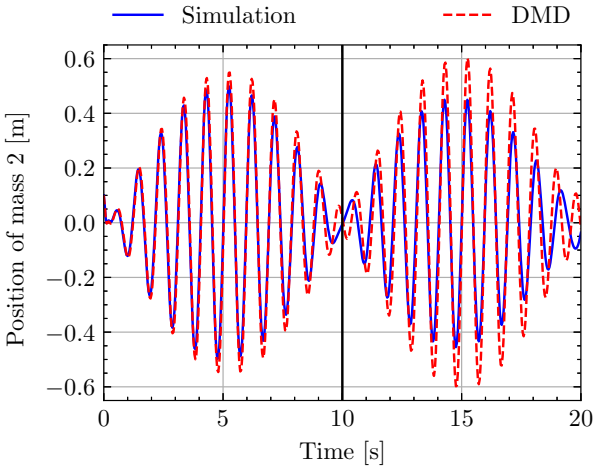


Figure 4.2: *Simulated solution (solid blue) compared to prediction (dashed red) of the position of mass 2 based on simulated data. Data up to time* 10 *s (the solid black line) was used for training.*

Even though the simulated data only deviates approximately 10^{-3} m from the analytical data in each step, the predictions are significantly worse compared to the ones when the data was generated by the analytical expression. The behaviour can be explained by studying the linear consistency of the data matrices. Recall Ex. 4.1.1, it is not surprising that X, Y constructed using the analytical expression are linearly consistent, resulting in perfect predictions. For the simulated data matrices we have $\mathcal{N}(X) \cap \mathcal{N}(Y) = \{\mathbf{0}\}$, meaning that we have no guarantee that $\hat{A}$ coincides with the true system matrix. Worth noting is that despite not having linearly consistent data the algorithm is still able to capture most of the behaviour of the system. It seems like the solution is somewhat unstable and that some high frequency behaviour is missed. The prediction errors are $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx \|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2 \approx 2.0$, where $\mathbf{x}_2^{sim}$ is the simulated solution for the position of the second mass, which is significantly higher than what was received in Ex. 4.1.1.

Example 4.3.3 (Impact of the sampling frequency)**.** According to the Nyquist-Shannon sampling theorem, we should be able to capture the behaviour of a system if it is sampled at a frequency of $2f_{max}$, called the Nyquist frequency f_N, where f_{max} is the highest frequency of the system. In practice however, a common guideline is that one should sample at least 10 times faster than f_{max}. If the total simulation time is constant, then increasing the number of snapshots gives an increased sampling frequency f_s. If we ignore the fact that the data matrices might not be linearly consistent, then we expect that f_s higher than $f_N = 2f_{max} = 2f_2 = 8.9$ Hz should be enough to capture the behaviour of the system. Moreover, we expect that the prediction error will decrease as f_s is increased.

Fig. 4.3a and 4.3b illustrate the difference in prediction performance between f_s equal to 10.2 Hz and 50.1 Hz respectively. A comparison can also be made to Fig. 4.2, where data has been sampled with a frequency of 25 Hz.

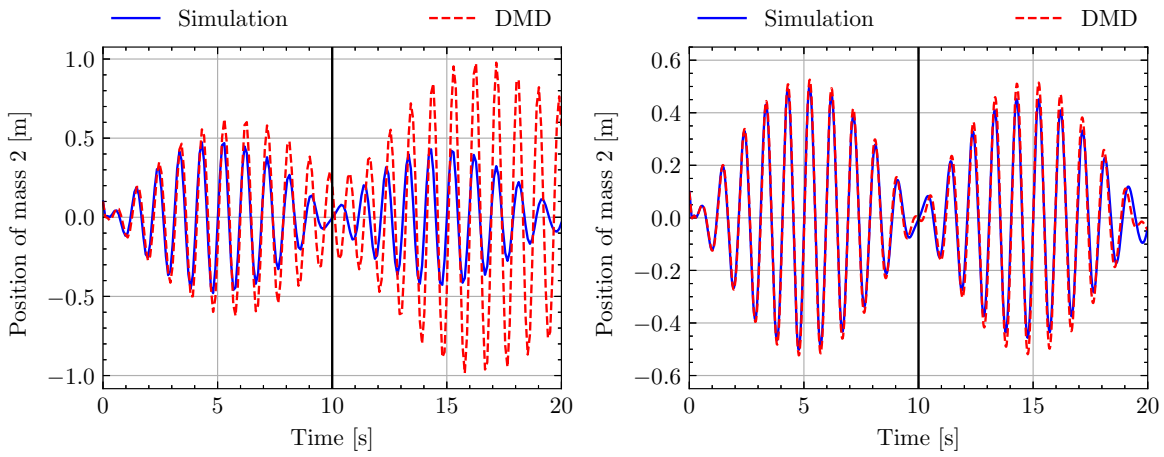


Figure 4.3: *Prediction of the position of mass 2 in red dashed lines and simulated solution in solid blue when using (4.3b)* 102 *and (4.3b)* 501 *snapshots.*

Even though all figures are based on data sampled with a frequency higher than f_N there are clear improvements in the performance of the DMD predictions when f_s is increased. More specifically, the prediction errors in Fig. 4.3a and 4.3b are $\|\hat{x}_2 - x_2^{an}\|_2 \approx 4.3$ and $\|\hat{x}_2 - x_2^{an}\|_2 \approx 1.4$ respectively. Perhaps the most noteworthy observation is that the amplitude of the solution in Fig. 4.3b is not increasing, unlike the solutions in the two other mentioned figures. This observation is not obvious from Fig. 4.3b, but upon further inspection over a longer time period it becomes clear that the oscillation is in fact damped.

Example 4.3.4 (DMD with time delay embedding)**.** As seen in Ex. 4.3.2, the concept of linear consistency is essential in order to achieve accurate predictions. We now present an example of how data that is not linearly consistent can be made linearly consistent using time delay embedding. This means that we are able to find a linear relation between X, Y even though they originate from a system where $Y \approx AX$ for some matrix A.

Consider the same setup as in Ex. 4.3.2 with data generated from simulations with the same solver. However, instead of constructing X, Y according to Eq. (4.1.3), time delay embedding, see Rem. 4.1.4, with $q = 2$ is utilized. This results in the data matrices

$$X = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_{m-1} \\ \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_m \end{bmatrix}, \quad Y = \begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_{m-1} \\ \mathbf{y}_2 & \mathbf{y}_3 & \cdots & \mathbf{y}_m \end{bmatrix} \tag{4.3.1}$$

and in this case $\hat{A} \in \mathbb{R}^{10\times 10}$ with

$$X_k = \begin{bmatrix} x_{1,k} & \dot{x}_{1,k} & x_{2,k} & \dot{x}_{2,k} & u_k & x_{1,k+1} & \dot{x}_{1,k+1} & x_{2,k+1} & \dot{x}_{2,k+1} & u_{k+1} \end{bmatrix}^T, \tag{4.3.2}$$

where X_k denotes the k:th column of X.

Performing DMD on X, Y and predictions according to Alg. 1 yields the result shown in Fig. 4.4.

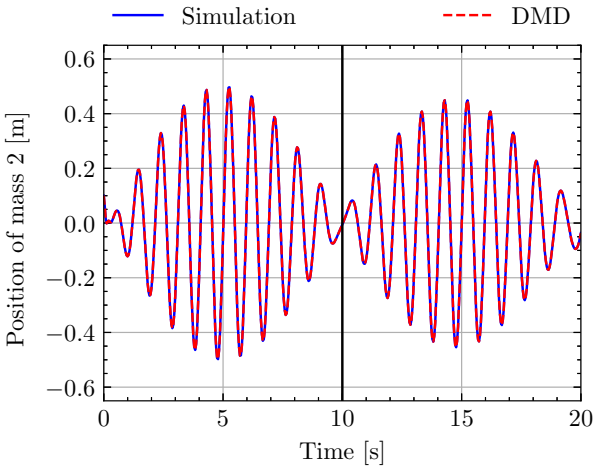


Figure 4.4: Improved predictions of the position of mass 2 compared to Fig. 4.2.

This result is clearly better than the corresponding result from Fig. 4.2. The predictions almost perfectly follow the simulation with the prediction errors $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx 10^{-5}$ and $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2 \approx 10^{-1}$. In conclusion, by only adding one more snapshot to Ex. 4.3.2 and arranging the data matrices as in Eq. (4.3.1), the predictions are significantly improved. More specifically, the predictions are almost as good as the simulated solution.

Example 4.3.5 (Minimum number of snapshots with analytical data)**.** Let us discuss the number of snapshots needed to construct $\hat{A}$ when using linearly consistent data. Suppose that we have five states, i.e. $\mathbf{x}_k \in \mathbb{R}^5$ and $X \in \mathbb{R}^{5\times m}$. There can be maximally five linearly independent column vectors in X so having $m > 5$ does not really add any additional information to X. We therefore expect six snapshots to be enough to be able to find the correct relation between X and Y, remember that $m+1$ snapshots are needed for the construction of X and Y. If time delay embeddings are used with $q = 2$ there are instead 10 states and we expect that 11 snapshots are needed to find the correct relation between X and Y.

Fig. 4.5 shows the result of DMD method when using analytical data and only using these minimum number of snapshots. The predictions are as expected well comparable with the analytical solution. This is also highlighted in the prediction errors, which are around 10^{-10} in both figures.

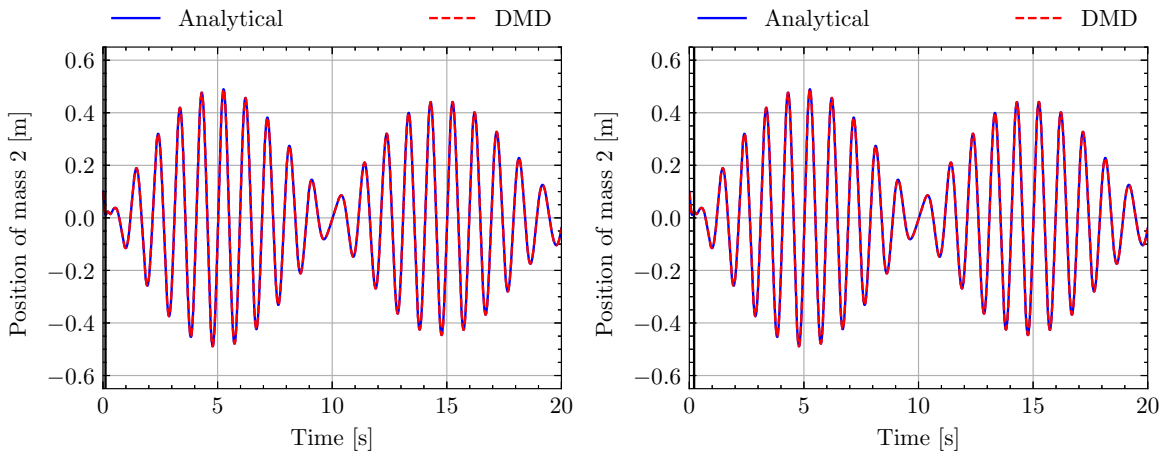


Figure 4.5: *Prediction of the position of mass 2 (red dashed line) and analytical solution (solid blue line) when using 6 snapshots and $q = 1$ (4.5a) and 11 snapshots and $q = 2$ (4.5b).*

Example 4.3.6 (Minimum number of snapshots with simulated data)**.** Consider the same setup as in the previous example, but the data is now generated from simulations. From Ex. 4.3.2 we know that the data matrices X, Y constructed from simulated data are not perfectly linearly consistent. This means that we do not expect the same result as in the previous example when the theoretical minimum number of snapshots are used.

Fig. 4.6 shows the result when using 6 snapshots with $q = 1$ and 11 snapshots with $q = 2$.

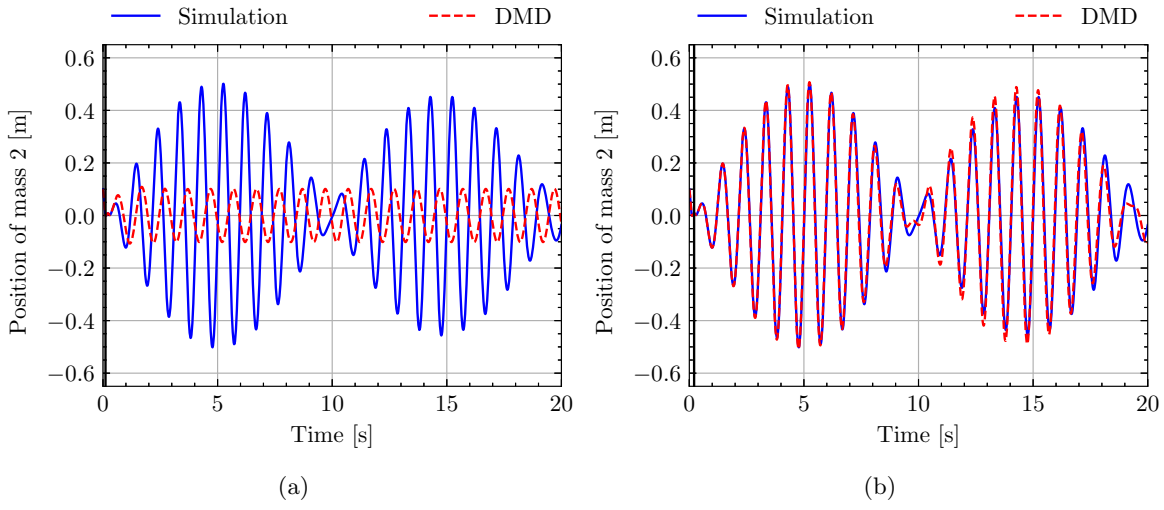


Figure 4.6: *Prediction of the position of mass 2 (red dashed line) and simulated solution (solid blue line) when using 6 snapshots with $q = 1$ (4.6a) and 11 snapshots with $q = 2$ (4.6b).*

When $q = 1$ the DMD method is not able to successfully predict the system from only 6 snapshots. The prediction error is $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx \|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2 \approx 10$. With $q = 2$ and 11 snapshots the DMD

method found a good representation of $\hat{A}$ with a prediction error $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx \|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2 \approx 1$. One interesting thing to note is that the prediction error continues to decrease when more snapshots are used, which can be seen by comparing to the result in Fig. 4.4. It is also worth mentioning that using 11 snapshots with $q = 1$ did not improve that result. To conclude, it seems like time delay embeddings make it possible to find a good $\hat{A}$ from relatively few snapshots.

Example 4.3.7 (DMD of data from a non-linear system)**.** Consider the non-linear version of the damped dual mass system, i.e. the system described in Sect. 2.1.2. As input a sine wave with amplitude 30 and frequency 0.1 Hz was used. We do not expect DMD to perform well on this system, since it is non-linear. Fig. 4.7 shows the predictions made by the DMD algorithm. The prediction lacks precision but manages to capture the dominant frequency and an amplitude that is close to the simulated solution. The received prediction error is $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx 1.6$. Note that no time delay embeddings have been used and we used 2000 snapshots for training.

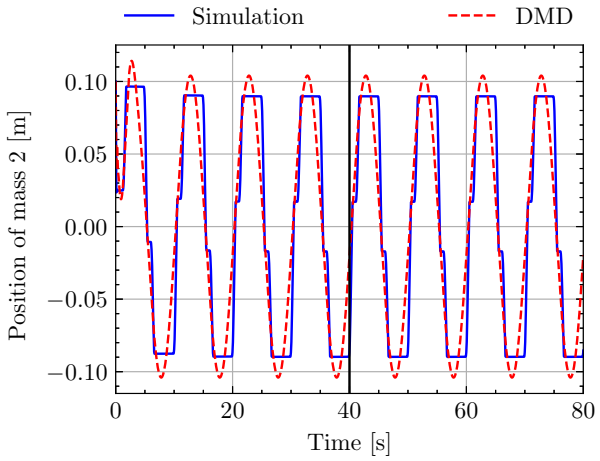
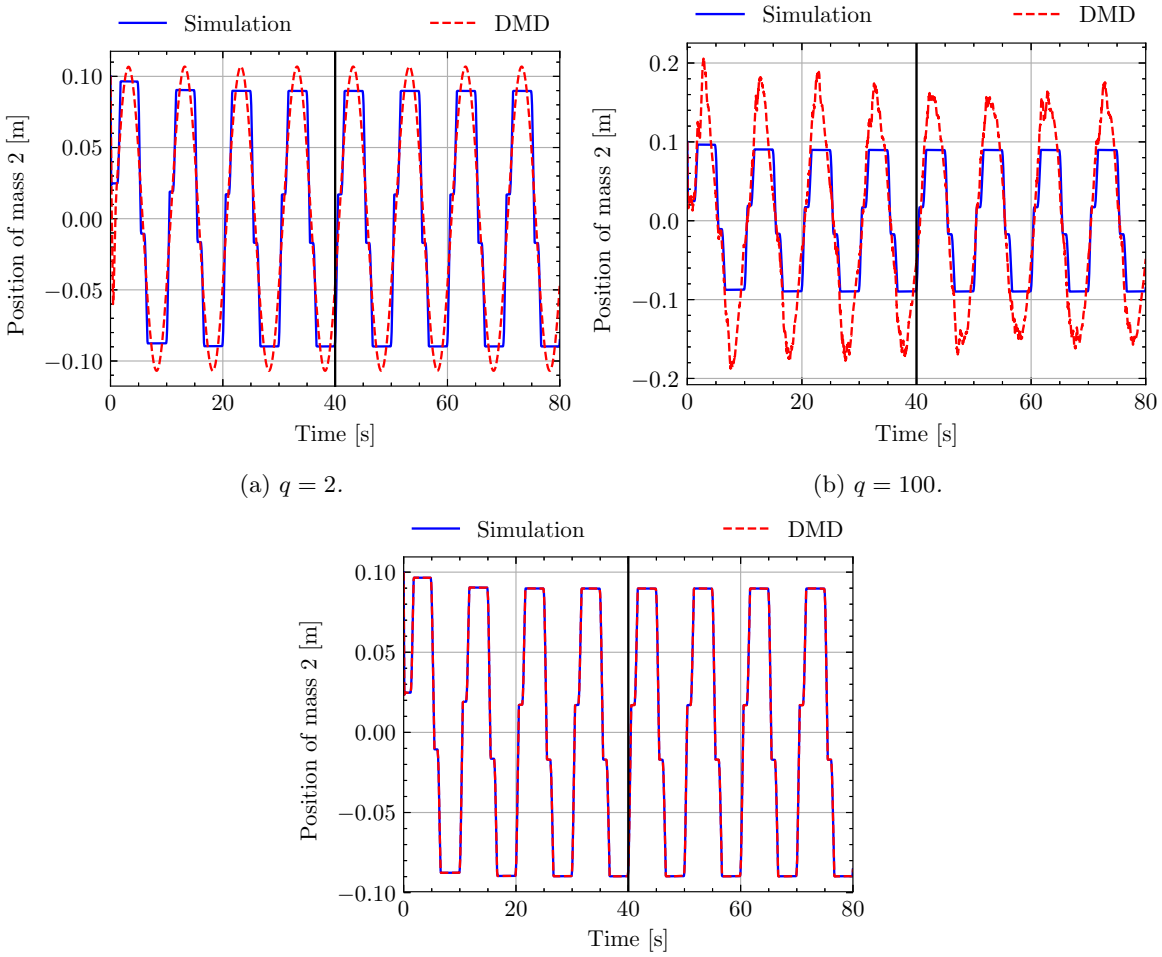


Figure 4.7: *Prediction of the position of mass 2 (red dashed line) and simulated solution (solid blue line).*

Example 4.3.8 (DMD with time delay embeddings in a non-linear setting)**.** A continuation of the previous example is to test if time delay embeddings can improve the result. For small values of q we do not expect any big improvements over Fig. 4.7 since the system is not approximately linear. If q is allowed to get large, say $q = 900$, that might change, since a lot of historical data will then be used to make a prediction about the next step. Fig. 4.8 shows the results for three different values of q.



(a) $q = 2$.

(b) $q = 100$.

(c) $q = 900$.

Figure 4.8: *Prediction of the position of mass 2 (red dashed line) and simulated solution (solid blue line) with different amounts of time delay embeddings.*

As expected, $q = 2$ did not significantly change the result. For $q = 100$ the result looks quite different compared to $q = 1$ with large overshoots but closer fit where the slope is steep. The really interesting result however is for $q = 900$. Here the DMD algorithm predicts the system accurately with a prediction error of only $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx 10^{-3}$.

Chapter 5

Koopman Spectral Analysis

It is not clear that the previously described DMD algorithm is a usable method for analysing a non-linear dynamical system. However, it turns out that DMD is closely related to Koopman spectral analysis. Using the Koopman operator one can transform a non-linear dynamical system into a linear, but infinite-dimensional, system through the usage of observable functions. The major issue with this is that we can only guarantee that this holds if we use a set of infinitely many observable functions. The remedy to this is to introduce the concept of Koopman invariant subspaces. By requiring the set of observables to span such a finite-dimensional subspace, the transformed system becomes both linear and finite-dimensional.

5.1 Koopman Operator Theory

The Koopman operator is defined for continuous and discrete dynamical systems. In this thesis we only consider the discrete case, and initially restrict to systems without input. Consider the discrete dynamical system in Eq. (1.2.2). For simplicity assume that $\mathbf{f}$ is an autonomous time-independent map, i.e.

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k), \tag{5.1.1}$$

where θ has been excluded compared to Eq. (1.2.2).

We now give a definition of the Koopman operator [24].

Definition 5.1.1 (Koopman operator)**.** Given any function $\mathbf{f} : \mathcal{M} \to \mathcal{M}$, the *Koopman operator* $\mathcal{K}$ is defined as an infinite-dimensional operator that maps any scalar function $g : \mathcal{M} \to \mathbb{R}$ to a new scalar function $\mathcal{K}g : \mathcal{M} \to \mathbb{R}$ given by

$$\mathcal{K}g(\mathbf{x}) = g(\mathbf{f}(\mathbf{x})). \tag{5.1.2}$$

Note that $\mathcal{K}$ is a linear operator since

$$\mathcal{K}(\alpha g_1 + \beta g_2)(\mathbf{x}) = (\alpha g_1 + \beta g_2)(\mathbf{f}(\mathbf{x})) = \alpha g_1(\mathbf{f}(\mathbf{x})) + \beta g_2(\mathbf{f}(\mathbf{x})) = \alpha \mathcal{K} g_1(\mathbf{x}) + \beta \mathcal{K} g_2(\mathbf{x}) \tag{5.1.3}$$

for any functions $g_1, g_2 : \mathcal{M} \to \mathbb{R}$ and scalars α, β. This means that the finite-dimensional non-linear system can be fully characterized as an infinite-dimensional linear system. Due to the linearity of $\mathcal{K}$ it can be decomposed into eigenvalues $\lambda_j \in \mathbb{C}$ and eigenfunctions $\varphi_j : \mathcal{M} \to \mathbb{C}$ as

$$\mathcal{K}\varphi_j(\mathbf{x}) = \lambda_j \varphi_j(\mathbf{x}), \quad j = 1, 2, \ldots, \infty. \tag{5.1.4}$$

The functions g, called observables, will play a crucial part in the Koopman analysis, as will be seen shortly.

Example 5.1.1 (Observable functions)**.** An observable function can be any scalar-valued function of the states in $\mathbf{x}$. If we consider the damped dual mass system in Sect. 2.1, we could for instance have $g(\mathbf{x}) = x_1 + x_2$, $g(\mathbf{x}) = x_1^2$ or simply the identity mapping $g(\mathbf{x}) = \mathbf{x}$.

Suppose that g lies in the span of $\{\varphi_j\}_{j=1}^{\infty}$. Then, g can be expressed as

$$g(\mathbf{x}) = \sum_{j=1}^{\infty} \varphi_j(\mathbf{x}) c_j \tag{5.1.5}$$

for some scalars $c_j \in \mathbb{C}$. Applying $\mathcal{K}$ to Eq. 5.1.5, utilizing the linearity of the operator and Eq. (5.1.4) gives

$$\mathcal{K}g(\mathbf{x}) = \mathcal{K}\left(\sum_{j=1}^{\infty} \varphi_j(\mathbf{x}) c_j\right) = \sum_{j=1}^{\infty} \mathcal{K}\varphi_j(\mathbf{x}) c_j = \sum_{j=1}^{\infty} \lambda_j \varphi_j(\mathbf{x}) c_j. \tag{5.1.6}$$

From Eq. (5.1.1) and Def. 5.1.1 it therefore follows that

$$g(\mathbf{x}_{k+1}) = g(\mathbf{f}(\mathbf{x}_k)) = \mathcal{K}g(\mathbf{x}_k) = \sum_{j=1}^{\infty} \lambda_j \varphi_j(\mathbf{x}_k) c_j. \tag{5.1.7}$$

Suppose that we have a vector-valued observable function $\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) & g_2(\mathbf{x}) & \dots & g_n(\mathbf{x}) \end{bmatrix}^T \in \mathbb{R}^n$ and that each component g_i lies in the span of $\{\varphi_j\}_{j=1}^{\infty}$. Then, these observables can be expressed as

$$\mathbf{g}(\mathbf{x}) = \sum_{j=1}^{\infty} \varphi_j(\mathbf{x}) \mathbf{v}_j \tag{5.1.8}$$

for some vectors $\mathbf{v}_j \in \mathbb{C}^n$. Later, $\mathbf{v}_j$ will denote the Koopman modes. Applying $\mathcal{K}$ to Eq. (5.1.8), utilizing the linearity of the operator and Eq. (5.1.4) give

$$\mathcal{K}\mathbf{g}(\mathbf{x}) = \mathcal{K}\left(\sum_{j=1}^{\infty} \varphi_j(\mathbf{x}) \mathbf{v}_j\right) = \sum_{j=1}^{\infty} \mathcal{K}\varphi_j(\mathbf{x}) \mathbf{v}_j = \sum_{j=1}^{\infty} \lambda_j \varphi_j(\mathbf{x}) \mathbf{v}_j. \tag{5.1.9}$$

According to Def. 5.1.1 we may then write $\mathcal{K}\mathbf{g}(\mathbf{x}) = \mathbf{g}(\mathbf{f}(\mathbf{x}))$, which combined with Eq. (5.1.1) and (5.1.9), yield

$$\mathbf{g}(\mathbf{x}_{k+1}) = \mathbf{g}(\mathbf{f}(\mathbf{x}_k)) = \mathcal{K}\mathbf{g}(\mathbf{x}_k) = \sum_{j=1}^{\infty} \lambda_j \varphi_j(\mathbf{x}_k) \mathbf{v}_j. \tag{5.1.10}$$

Eq. (5.1.10) is an infinite sum describing how the observables $\mathbf{g}(\mathbf{x})$ evolve over distinct time steps.

The goal is to find a representation of $\mathcal{K}$ on a finite-dimensional subspace. In accordance with Takeishi *et al.* [26], the concept of Koopman invariant subspaces is introduced in order to assist in this representation.

Definition 5.1.2 (Koopman invariant subspace)**.** Let $\mathcal{G}$ be a finite-dimensional subset of the infinite-dimensional set of functions $g : \mathcal{M} \to \mathbb{R}$. Furthermore, let $\mathcal{G}$ be invariant under $\mathcal{K}$, i.e. $\mathcal{K}g \in \mathcal{G}$, $\forall g \in \mathcal{G}$. Then, $\mathcal{G}$ is called a *Koopman invariant subspace*.

Remark 5.1.1. If $\mathcal{G}$ is spanned by a set of n functions, we can represent $\mathcal{K}$ restricted to $\mathcal{G}$ by a finite-dimensional linear operator $\tilde{\mathcal{K}} : \mathcal{G} \to \mathcal{G}$, which in turn can represented by a matrix $K \in \mathbb{R}^{n\times n}$.

In the following theorem from [26], we show that a representation of K can be computed from n square-integrable observables if and only if they span a Koopman invariant subspace. This guarantees a linear evolution of the observables and allow us to reduce Eq. (5.1.2) from being infinite-dimensional to being n-dimensional.

Theorem 5.1.1. *Consider a set of observables $\{g_i\}_{i=1}^n \in L^2(\mathcal{M})$ and define the matrix $G \in \mathbb{R}^{n\times n}$ as*

$$G = (\mathbf{g} \circ \mathbf{f}, \mathbf{g})(\mathbf{g}, \mathbf{g})^\dagger, \tag{5.1.11}$$

where $(\mathbf{f}, \mathbf{g}) = \int_{\mathcal{M}} \mathbf{f}(\mathbf{x})\mathbf{g}(\mathbf{x})^T d\mathbf{x}$. Then,

$$G\mathbf{g}(\mathbf{x}) = \mathbf{g}(\mathbf{f}(\mathbf{x})), \quad \forall \mathbf{x} \in \mathcal{M}, \tag{5.1.12}$$

i.e. K is represented by G, if and only if $\{g_i\}_{i=1}^n$ spans a Koopman invariant subspace.

Proof. Suppose $\mathbf{g}(\mathbf{f}(\mathbf{x})) = G\mathbf{g}(\mathbf{x})$, $\forall \mathbf{x} \in \mathcal{M}$. For any $\tilde{\mathbf{g}} \in \text{span}(\{g_i\}_{i=1}^n)$ it holds that $\tilde{\mathbf{g}} = \sum_{i=1}^n a_i g_i \in \mathbb{R}^n$, for some $a_i \in \mathbb{R}$. Thus

$$\mathcal{K}\tilde{\mathbf{g}} = \mathcal{K}\sum_{i=1}^n a_i g_i = \sum_{i=1}^n a_i \mathcal{K} g_i = \sum_{i=1}^n a_i (g_i \circ \mathbf{f}) = \sum_{i=1}^n a_i \left(\sum_{j=1}^n G_{i,j} g_j \right), \tag{5.1.13}$$

where $G_{i,j}$ denotes the (i,j)-element of G and we used that $g_i \circ \mathbf{f} = \sum_j G_{i,j} g_j$ in the last equality. Since the order of summation does not matter we write this as

$$\sum_{i=1}^n a_i \left(\sum_{j=1}^n G_{i,j} g_j \right) = \sum_{j=1}^n \left(\sum_{i=1}^n a_i G_{i,j} \right) g_j \in \text{span}(\{g_i\}_{i=1}^n), \tag{5.1.14}$$

so $\{g_i\}_{i=1}^n$ spans a Koopman invariant subspace.

Let us now assume that $\{g_1, \ldots, g_n\} \in \mathrm{L}^2(\mathcal{M})$ and spans a Koopman invariant subspace. Then, there exists a finite-dimensional linear operator $\tilde{\mathcal{K}}$ such that $\tilde{\mathcal{K}}\mathbf{g}(\mathbf{x}) = \mathbf{g}(\mathbf{f}(\mathbf{x}))$ $\forall \mathbf{x} \in \mathcal{M}$. This gives that $\tilde{\mathcal{K}}$ fulfils

$$\tilde{\mathcal{K}}(\mathbf{g}, \mathbf{g}) = (\tilde{\mathcal{K}}\mathbf{g}, \mathbf{g}) = (\mathbf{g} \circ \mathbf{f}, \mathbf{g}). \tag{5.1.15}$$

From the definition of G in Eq. (5.1.11) it follows that G also satisfies Eq. (5.1.15), since

$$G(\mathbf{g}, \mathbf{g}) = (\mathbf{g} \circ \mathbf{f}, \mathbf{g})(\mathbf{g}, \mathbf{g})^\dagger (\mathbf{g}, \mathbf{g}) = (\mathbf{g} \circ \mathbf{f}, \mathbf{g}), \tag{5.1.16}$$

i.e. G is one instance of $\tilde{\mathcal{K}}$. □

To summarize Thm. 5.1.1, the Koopman operator can be represented by an n-dimensional matrix G, if the set of observable functions $\{g_1, \ldots, g_n\} \in \mathrm{L}^2(\mathcal{M})$ and spans a Koopman invariant subspace. Furthermore, by introducing the matrices $X, Y \in \mathbb{R}^{n\times m}$ as

$$X = \begin{bmatrix} \mathbf{g}(\mathbf{x}_1) & \cdots & \mathbf{g}(\mathbf{x}_m) \end{bmatrix}, \quad Y = \begin{bmatrix} \mathbf{g}(\mathbf{f}(\mathbf{x}_1)) & \cdots & \mathbf{g}(\mathbf{f}(\mathbf{x}_m)) \end{bmatrix}, \tag{5.1.17}$$

Thm. 5.1.1 and Eq. 5.1.2 give the linear relation between X and Y as

$$GX = Y. \tag{5.1.18}$$

An illustration of how the observables transform the states is shown in Fig. 5.1, where it is assumed that the set of observables span a Koopman invariant subspace.

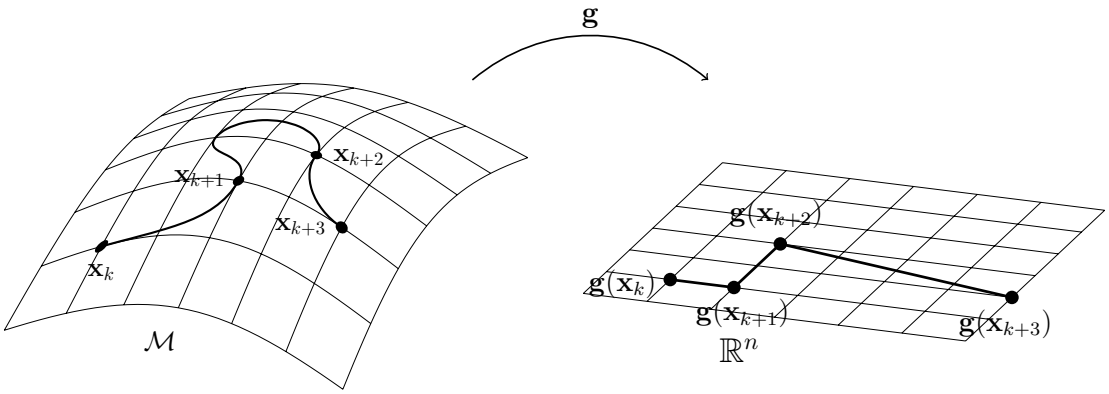


Figure 5.1: *Illustration of how the vector-valued observable function* $\mathbf{g}$ *transforms the states, which evolve non-linearly in time on* $\mathcal{M}$*, to evolve linearly in time in* $\mathbb{R}^n$*, given that the set of observables span a Koopman invariant subspace.*

The next theorem from Takeishi *et al.* [26] presents how an approximation of G can be computed from the matrices X, Y. For a complete proof the reader is referred to [26].

Theorem 5.1.2. *Define* X, Y *according to Eq.* (5.1.17) *and assume that Birkhoff's Ergodic Theorem holds, see [6]. If rank*$(X) = n$*, then* $\hat{A} \in \mathbb{R}^{n\times n}$ *given by*

$$\hat{A} = YX^{\dagger} \tag{5.1.19}$$

converges to G *defined in Eq.* (5.1.11) *as the number of snapshots* $m \to \infty$*.*

Thm. 5.1.2 states how an approximation of G can be found from the data matrices X, Y, given that X has full rank. This matrix $\hat{A}$ is calculated in the same way as $\hat{A}$ in Def. 4.1.1.

By combining Thm. 5.1.1 and 5.1.2, we arrive at the result that if $\{g_i\}_{i=1}^n$ spans a Koopman invariant subspace then Eq. (5.1.12) holds, where we may set $K = G = \hat{A}$. In other words, for sufficiently large m Eq. (5.1.18) turns into

$$\hat{A}X = Y. \tag{5.1.20}$$

The observable functions can therefore be interpreted as a mapping from a domain where the system is non-linear to a new domain where the system is linear.

Analogously to Ch. 4 we may perform a DMD on $\hat{A}$. The computation of the DMD of data matrices X, Y defined as in Eq. (5.1.17) is referred to as extended dynamic mode decomposition (EDMD) [12]. If $\hat{A}$ has a full set of right eigenvectors $\mathbf{v}_i \in \mathbb{C}^n$ with corresponding eigenvalues $\lambda_i \in \mathbb{C}$ we may write

$$\mathbf{g}(\mathbf{x}_k) = \sum_{i=1}^{n} \langle \mathbf{w}_i, \mathbf{g}(\mathbf{x}_k)\rangle \mathbf{v}_i. \tag{5.1.21}$$

where $\mathbf{w}_i \in \mathbb{C}^n$ are the left eigenvectors of $\hat{A}$. Introduce $\mathbf{y}_k = \mathbf{g}(\mathbf{f}(\mathbf{x}_k)) = \mathbf{g}(\mathbf{x}_{k+1}) \in \mathbb{R}^n$ and let $\hat{\mathbf{y}}_k$ denote the prediction of $\mathbf{y}_k$. Applying $\hat{A}$ to Eq. 5.1.21 yields

$$\hat{\mathbf{y}}_k = \hat{A}\mathbf{g}(\mathbf{x}_k) = \sum_{i=1}^{n} \lambda_i \langle \mathbf{w}_i, \mathbf{g}(\mathbf{x}_k)\rangle \mathbf{v}_i. \tag{5.1.22}$$

Comparing with Eq. 4.1.10 reveals that the DMD modes and eigenvalues coincide with the modes and eigenvalues of the Koopman operator for sufficiently many snapshots m.

To summarize, if the system of interest is non-linear, then we may use Alg. 1 with a slight modification to capture its characteristics and make future predictions. More specifically, there are two differences in the algorithm. One, the construction of the data matrices (step 1) is done from the vector-valued observables instead of the states. Two, the prediction (step 6) no longer gives predictions of the states, but predictions of the observables.

In this section we have shown that DMD, due to its connection with Koopman spectral analysis, is a tool that can be used to extract characteristics of non-linear dynamical systems.

5.2 Connection to Dynamic Mode Decomposition

We now motivate why the method described in this chapter can be seen as an extension to the DMD method. As seen in a previous example, we may choose our observables such that each of them maps to a single state in $\mathbf{x}$, for instance $g_1(\mathbf{x}) = x_1$. Choosing such identity mapping observables for the damped dual mass system gives

$$\mathbf{g}(\mathbf{x}_k) = \begin{bmatrix} g_1(\mathbf{x}_k) \\ g_2(\mathbf{x}_k) \\ g_3(\mathbf{x}_k) \\ g_4(\mathbf{x}_k) \end{bmatrix} = \begin{bmatrix} x_{1,k} \\ \dot{x}_{1,k} \\ x_{2,k} \\ \dot{x}_{2,k} \end{bmatrix} = \mathbf{x}_k \tag{5.2.1}$$

and analogously

$$\mathbf{g}(\mathbf{f}(\mathbf{x}_k)) = \begin{bmatrix} g_1(\mathbf{f}(\mathbf{x}_k)) \\ g_2(\mathbf{f}(\mathbf{x}_k)) \\ g_3(\mathbf{f}(\mathbf{x}_k)) \\ g_4(\mathbf{f}(\mathbf{x}_k)) \end{bmatrix} = \begin{bmatrix} g_1(\mathbf{x}_{k+1}) \\ g_2(\mathbf{x}_{k+1}) \\ g_3(\mathbf{x}_{k+1}) \\ g_4(\mathbf{x}_{k+1}) \end{bmatrix} = \begin{bmatrix} x_{1,k+1} \\ \dot{x}_{1,k+1} \\ x_{2,k+1} \\ \dot{x}_{2,k+1} \end{bmatrix} = \mathbf{x}_{k+1}. \tag{5.2.2}$$

This results in the data matrices

$$\begin{aligned} X &= \begin{bmatrix} \mathbf{g}(\mathbf{x}_1) & \cdots & \mathbf{g}(\mathbf{x}_m) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_m \end{bmatrix} \\ Y &= \begin{bmatrix} \mathbf{g}(\mathbf{f}(\mathbf{x}_1)) & \cdots & \mathbf{g}(\mathbf{f}(\mathbf{x}_m)) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_2 & \cdots & \mathbf{x}_{m+1} \end{bmatrix}, \end{aligned} \tag{5.2.3}$$

which are the same as in the definition of DMD (Def. 4.1.1), just with sequential data. Consequently, Eq. (5.1.21) and (5.1.22) reduce to the corresponding equations in the DMD chapter (Eq. (4.1.9) and (4.1.10)) and hence this example will equal Ex. 4.1.1. The conclusion is that the method described in the DMD section is nothing else but a special case of the Koopman method when identity mapping observables are used. In this linear setting, the Koopman modes will equal the DMD modes, i.e. the eigenvectors of $\hat{A}$.

5.3 Examples

We now present a few examples that show the capabilities of the Koopman method and how it compares to the DMD method. Especially the importance of the observable functions will be highlighted.

Example 5.3.1 (Connection between DMD and Koopman spectral analysis)**.** Consider the setup from Ex. 4.3.1, i.e. the damped dual mass system with a sine input force sampled 25 times per second for 10 seconds. We want to apply Koopman spectral analysis to this problem and since it is linear in $\mathbf{x}$ we choose $\mathbf{g}(\mathbf{x}) = \mathbf{x}$. The result is expected to be identical to the one where DMD is used, see Ex. 4.3.1.

In Fig. 5.2 this is exactly what we find. The computations also reveal that the difference between the solutions when using the Koopman and DMD method is identically zero. This is also confirmed by the prediction error $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2 \approx 10^{-12}$, which is identical to the prediction error received when using DMD.

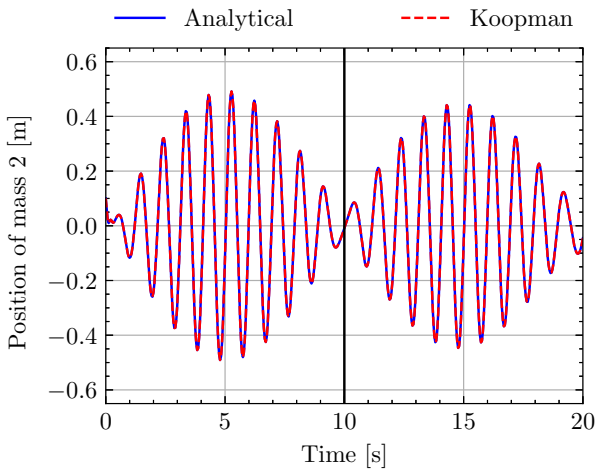


Figure 5.2: *Analytical solution (solid blue) compared to prediction (dashed red) of the position of mass 2 based on analytical data. Data up to time* 10 *(the solid black line) was used for training.*

Example 5.3.2 (The impact of a rank-deficient X)**.** In Thm. 5.1.2, one of the requirements to guarantee that $\hat{A}$ converges to G was that $X \in \mathbb{R}^{n\times m}$ should have full rank. We now present a closer look at what happens when X is rank-deficient, i.e. when $\text{rank}(X) < n$ (assuming $n < m$).

Let $\tilde{X} \in \mathbb{R}^{\tilde{n}\times m}$ be a matrix, where $\tilde{n} \geq n$. The first n rows of $\tilde{X}$ are the rows from X and the remaining $\tilde{n} - n$ rows are duplicates of some of the already existing ones. Let $\tilde{Y}$ be constructed analogously to $\tilde{X}$. It holds that $\text{rank}(X) = \text{rank}(\tilde{X}) = n$ and that $\tilde{X}$ does not have full rank if $\tilde{n} > n$.

Consider again the test setup from Ex. 4.1.1, i.e. the damped dual mass system with a sine input signal where the data has been generated analytically. Set $\mathbf{g}(\mathbf{x}) = \mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^5$, which gives $n = 5$. We use 250 snapshots, i.e. $m = 250$. Since $m \geq 5$ we have $\text{rank}(X) = 5$ and therefore $\text{rank}(\tilde{X}) = 5$. Tab. 5.1 presents the prediction error received for some values of $\tilde{n}$.

Table 5.1: *Prediction error for the position of the second mass for some values of* $\tilde{n}$*. For* $\tilde{n} = 8$ *the prediction error was too large to be computed. Note that rank*$(\tilde{X}) = 5$ *for all cases.*

$\tilde{n}$	5	6	7	8
$\tilde{n} - \text{rank}(\tilde{X})$	0	1	2	3
$\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{an}\|_2$	$2.4 \cdot 10^{-12}$	$5.4 \cdot 10^{-12}$	$5.1 \cdot 10^{-10}$	-

For $\tilde{n} = 5, 6, 7$ there are small but measurable differences in the prediction errors. For $\tilde{n} = 8$ the predictions become unstable and go towards infinity, hence no given prediction error in Tab. 5.1. One interesting thing to note is that it was possible to receive accurate predictions even when $\tilde{X}$ was rank-deficient. This result does not violate Thm. 5.1.2, since the theorem does not say anything about what happens when X does not have full rank.

Example 5.3.3 (Observables that span a Koopman invariant subspace)**.** Consider a dynamical system in $\mathbb{R}^2$

$$\begin{aligned} \dot{x}_1 &= \rho x_1 \\ \dot{x}_2 &= \xi(x_2 - x_1^2), \end{aligned} \tag{5.3.1}$$

for some constants ρ and ξ. Let $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$ and introduce the following vector-valued observable function

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ g_3(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \end{bmatrix}. \tag{5.3.2}$$

For easier notation we introduce $\mathbf{y} = \mathbf{g}(\mathbf{x}) \in \mathbb{R}^3$, which gives

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 \\ y_3 &= x_1^2. \end{aligned} \tag{5.3.3}$$

Eq. (5.3.1) can now be rewritten in terms of $\mathbf{y}$ as

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{bmatrix} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ 2x_1\dot{x}_1 \end{bmatrix} = \begin{bmatrix} \rho x_1 \\ \xi(x_2 - x_1^2) \\ 2\rho x_1^2 \end{bmatrix} = \begin{bmatrix} \rho y_1 \\ \xi(y_2 - y_3) \\ 2\rho y_3 \end{bmatrix} = \begin{bmatrix} \rho & 0 & 0 \\ 0 & \xi & -\xi \\ 0 & 0 & 2\rho \end{bmatrix} \mathbf{y} = A\mathbf{y}, \tag{5.3.4}$$

which is a linear system in $\mathbf{y}$. To summarize, by choosing the observable function according to Eq. (5.3.2) the system in Eq. (5.3.1) is transformed from a non-linear system into a linear system on the form of Eq. (5.3.4). This means that $\{g_1, g_2, g_3\}$ spans a Koopman invariant subspace.

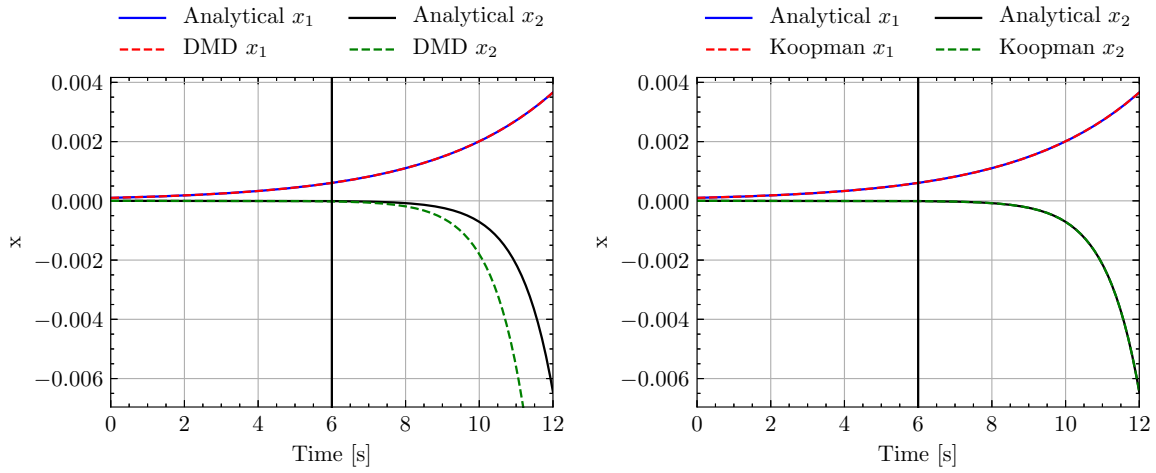
Example 5.3.4 (Comparison between DMD and the Koopman method in a non-linear setting)**.** We now illustrate the difference between the solutions from the DMD and the Koopman method on a non-linear dynamical system. More specifically, we again consider the planar ODE system in Eq. (5.3.1), which is linear in x_1 and non-linear in x_2. We therefore expect the DMD to perform well on predicting x_1 but not on x_2. We expect the Koopman method to predict both states well since, as seen in Ex. 5.3.3, we were able to make the system linear by introducing $\mathbf{g}(\mathbf{x}) = \begin{bmatrix} x_1 & x_2 & x_1^2 \end{bmatrix}^T$.

By using this vector-valued observable function $\mathbf{g}(\mathbf{x})$, we are able to solve the system for x_1 and x_2 analytically and use this as data for the two methods. This is done analogously to Sect. 2.1 by reformulating Eq. (5.3.4) to the discrete case as $\mathbf{y}_{k+1} = e^{A\Delta t}\mathbf{y}_k$ where k denotes the time index such that $t = k\Delta t$ for some time step Δt. The data was generated using $\rho = 0.3$, $\xi = 1.1$, $\Delta t = 0.01$ s and the initial value was set to $\mathbf{x}_0 = \begin{bmatrix} 10^{-4} & 10^{-8} \end{bmatrix}^T$.

Both methods use the same amount of data, namely data up to $t = 6$ s, and predicts both states from $t = 0$ s to $t = 12$ s. The predictions are shown in Fig. 5.3 and the errors, when compared to the analytical solution, in Tab. 5.2. In Fig. 5.3a, we can see, as expected, that the DMD successfully predicts the linear state x_1, whereas it has trouble with the non-linear state x_2. When the Koopman method was used, we can see in Fig. 5.3b that it performs in line with our expectations, namely that it predicts both states well. The prediction errors in Tab. 5.2 also confirms our hypotheses since the error of the prediction of x_1 is around 10^{-12} for both methods and that the error for x_2 is approximately 10^{-2} and 10^{-13} for DMD and Koopman respectively. In conclusion, this example illustrates the limitations of DMD and how the Koopman theory is able to resolve these. Worth noting is that we manage to get these results from the Koopman method only because we were able to find suitable observable functions. In the following example, we will see how this choice affects the performance of the Koopman method.

Table 5.2: *Prediction errors when using the DMD and Koopman method.*

Method	DMD	Koopman
$\|\hat{x}_1 - x_1^{an}\|_2$	$6.5 \cdot 10^{-13}$	$7.4 \cdot 10^{-12}$
$\|\hat{x}_2 - x_2^{an}\|_2$	$7.3 \cdot 10^{-2}$	$5.0 \cdot 10^{-13}$



(a) (b)

Figure 5.3: *Prediction of the states x_1 and x_2 when using the DMD (5.3a) and the Koopman method (5.3b) in dashed red and green. The solid blue and black lines show the analytical solution for the two states respectively.*

Example 5.3.5 (Observables that do not span a Koopman invariant subspace)**.** Consider the system from Ex. 5.3.3 with the same parameters and initial value and let $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \in \mathbb{R}^2$. Introduce the following vector-valued observable function

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ g_3(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_2^2 \end{bmatrix}. \tag{5.3.5}$$

For easier notation we introduce $\mathbf{y} = \mathbf{g}(\mathbf{x}) \in \mathbb{R}^3$, which gives

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 \\ y_3 &= x_2^2. \end{aligned} \tag{5.3.6}$$

Eq. (5.3.1) can now be rewritten in terms of $\mathbf{y}$ as

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{bmatrix} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ 2x_2\dot{x}_2 \end{bmatrix} = \begin{bmatrix} \rho x_1 \\ \xi(x_2 - x_1^2) \\ 2\xi(x_2^2 - x_1^2 x_2) \end{bmatrix} = \begin{bmatrix} \rho y_1 \\ \xi(y_2 - y_1^2) \\ 2\xi(y_3 - y_1^2 y_2) \end{bmatrix}, \tag{5.3.7}$$

which is not a linear system in $\mathbf{y}$. Thus, we do not expect the predictions of the x_2 to be accurate, which also is confirmed by the numerical result seen in Fig. 5.4. Comparing this with Fig. 5.3b it is clear that the choice of observables are important.

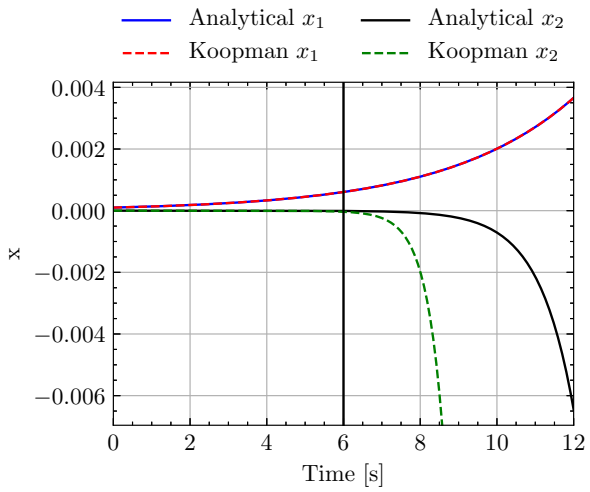
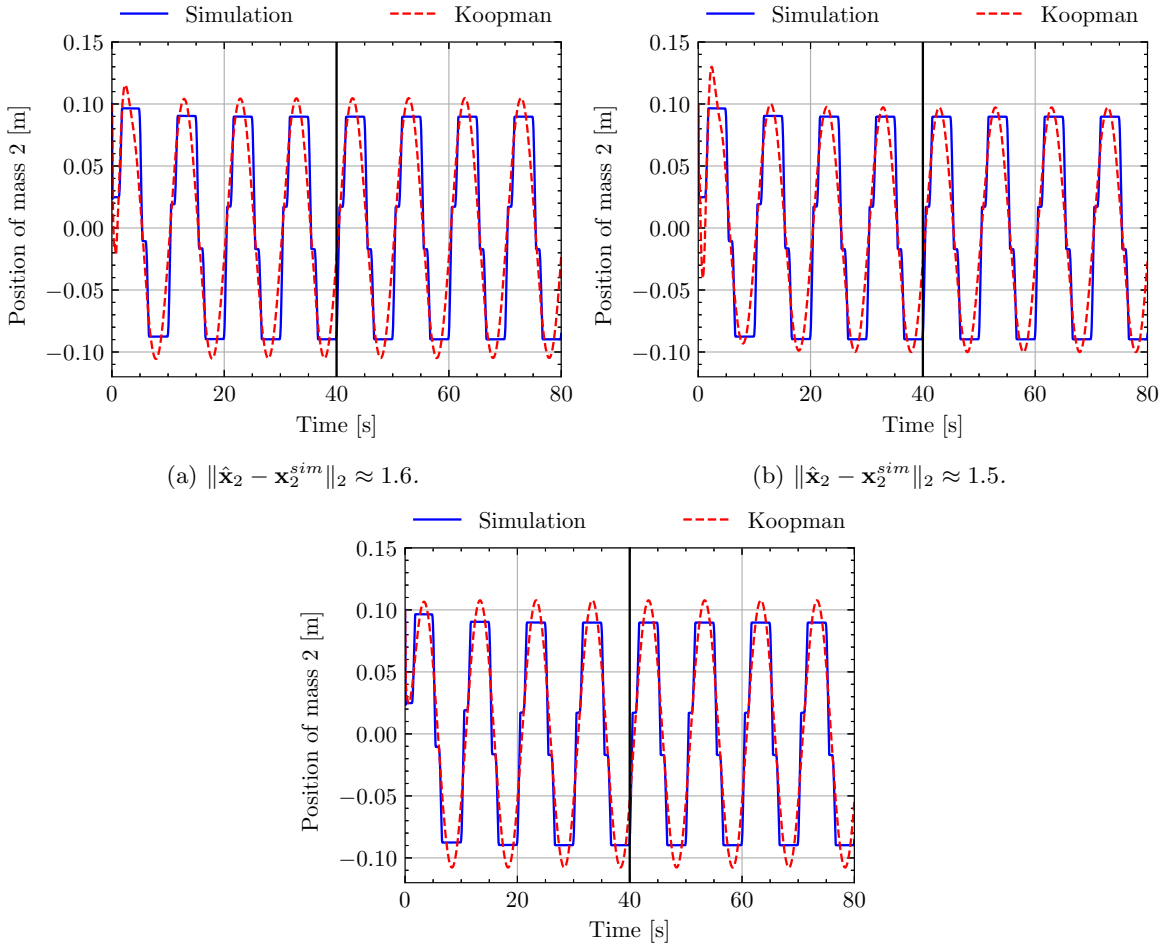


Figure 5.4: *Prediction of the states x_1 and x_2 when using the Koopman method in dashed red and green. The solid blue and black lines show the analytical solution for the two states respectively. The prediction errors are $\|x_1 - x_1^{an}\|_2 \approx 10^{-9}$ and $\|x_2 - x_2^{an}\|_2 \approx 10^2$.*

Example 5.3.6 (Manually choosing observables in a non-linear setting)**.** To illustrate how hard it is to choose the observable functions the setup from Ex. 4.3.7 was used. Three different observable functions were used, namely

$$\begin{aligned}
\mathbf{g}_1(\mathbf{x}) &= \begin{bmatrix} x_1 & v_1 & x_2 & v_2 & x_1^2 & x_2^2 & x_1x_2 & v_1^2 & v_2^2 & v_1v_2 & x_1v_1 & x_2v_2 \end{bmatrix}^T \\
\mathbf{g}_2(\mathbf{x}) &= \begin{bmatrix} x_1 & v_1 & x_2 & v_2 & \sin(x_1) & \sin(x_2) & \cos(x_1) & \cos(x_2) \end{bmatrix}^T \\
\mathbf{g}_3(\mathbf{x}) &= \begin{bmatrix} x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & x_1^3 & x_1^2x_2 & x_1x_2^2 & x_2^3 & x_1^4 & x_1^3x_2 & x_1^2x_2^2 & x_1x_2^3 & x_2^4 \end{bmatrix}^T .
\end{aligned} \tag{5.3.8}$$

Fig. 5.5 shows the results with the prediction error for each $\mathbf{g}$ in Eq. (5.3.8). The results are very similar to those of Ex. 4.3.7, so guessing observable functions was not a successful approach, which hardly comes as a surprise. We again point out that the observables have to span a Koopman invariant subspace in order for G to be a representation of K.



(a) $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx 1.6$.

(b) $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx 1.5$.

(c) $\|\hat{\mathbf{x}}_2 - \mathbf{x}_2^{sim}\|_2 \approx 1.1$.

Figure 5.5: *Prediction of the position of mass 2 (red dashed line) and simulated solution (solid blue line) when using the three different observable functions from Eq.* (5.3.8)

Chapter 6

Koopman Spectral Analysis with Neural Networks

In Ex. 5.3.3-5.3.5 we saw how crucial and non-trivial it is to find suitable observable functions, even for a simple non-linear dynamical system. When dealing with systems with thousands of highly non-linear equations this choice becomes even harder. In simple terms, the problem we are facing is to find n functions that fulfil two specific criteria. Firstly, that the functions are square-integrable. Secondly, the set of these functions should span a Koopman invariant subspace. See Thm. 5.1.1 for the motivation why this has to hold.

One way to find these functions is to create a parametrized general function. The parameters of this general function can be tuned by means of optimization such that the function satisfies these criteria. The question then arises which general function one should start with. Numerous works have shown that neural networks can be used to represent an approximation of wide set of different function classes. More specifically, a feedforward neural network with one hidden layer and a locally bounded non-constant activation function $\sigma : \mathbb{R} \to \mathbb{R}$, that is not a polynomial, can to any level of accuracy approximate any function in $L^p(\mu)$, where closeness is measured in the measure μ. We require the observable functions to be square-integrable, which implies $p = 2$. Admissible activation functions include e.g. the sigmoid function, hyperbolic tangent, rectified linear unit (ReLU) and leaky rectified linear unit (leaky ReLU). This result is often summarized in the Universal approximation theorem [17, 16, 19]. Noteworthy is that this theorem only tells us that there exists such a feedforward neural network but gives no guarantees that the optimization algorithm is able to find that network. It is also worth mentioning that research is being done to try to instead directly find a neural network representation of the eigenfunctions of the Koopman operator [20]. This is however not considered in this thesis.

We now give a coarse description of a feedforward neural network and how it can be utilized to find appropriate observable functions.

6.1 Feedforward Neural Networks

A feedforward neural network can be seen as a function $\mathbf{\Phi} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_L}$ that is composed of functions ϕ_i such that $\mathbf{\Phi}(\mathbf{x}) = \phi_L \circ \cdots \circ \phi_2 \circ \phi_1(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^{d_1}$. Each $\phi_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_{i+1}}$ represents a layer with N_i neurons. The i:th layer can be of any dimension as long as its input and output dimension coincide

with the corresponding dimensions of the adjacent layers ϕ_{i-1} and ϕ_{i+1}. The first layer, ϕ_1, is defined as the input layer and the last layer, ϕ_L, as the output layer. The layers in between are referred to as the hidden layers. The depth of the network is defined as the number of layers in the network, which in this case is L [11]. We define the size of the network as the total number of neurons $N = \sum_{i=1}^{L} N_i$. The goal with a neural network is to find an approximation of some function $\mathbf{\Phi}^*$. Consider $\mathcal{F}(N, L)$ as the set of all feedforward neural networks of size N and depth L. We then wish to find $\mathbf{\Phi}$ such that

$$\min_{\tilde{\mathbf{\Phi}} \in \mathcal{F}(N,L)} \|\tilde{\mathbf{\Phi}}(\cdot) - \mathbf{\Phi}^*(\cdot)\| \leq \varepsilon \tag{6.1.1}$$

for some small ε.

To choose the size, depth and the functions ϕ_i, i.e. the structure of the network, such that the network after being trained closely resembles $\mathbf{\Phi}^*$, is not trivial. Generally a combination of linear and non-linear functions are used to build up a neural network. This can be achieved by using affine functions $\phi(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^T\mathbf{w} + b$ with the input $\mathbf{x} \in \mathbb{R}^n$, the weights $\mathbf{w} \in \mathbb{R}^n$ and the bias $b \in \mathbb{R}$. Such a layer is often referred to as a fully connected or an affine layer. The weights and biases are called the parameters $\theta = \begin{bmatrix}\mathbf{w}^T & b\end{bmatrix}^T$ of the layer and are learned through training. It is common to initialize the elements of $\mathbf{w}$ from some uniform or normal distribution, whereas the elements of $\mathbf{b}$ are often initialized to zero. In contrast to the affine layers, the non-linear functions, called activation functions, have fixed parameters [11].

There are of course many other types of layers that can be used, but in this thesis we will only consider affine layers and activation functions.

6.2 Training a Neural Network

Consider a training example $(\mathbf{x}_j, \mathbf{y}_j)$, where $\mathbf{x}_j \in \mathbb{R}^{d_1}$ is the input to the network $\mathbf{\Phi}$ and $\mathbf{y}_j \in \mathbb{R}^{d_L}$ is the target. Let $\hat{\mathbf{y}}_j = \mathbf{\Phi}(\mathbf{x}_j)$ be the output of the network. Furthermore, consider $\mathbf{\Phi}$ only consisting of affine functions and activation functions, each with the parameters θ_i, i.e. $\mathbf{\Phi}(\mathbf{x}; \theta)$ where $\theta = \{\theta_1, \theta_2, \ldots, \theta_L\}$. The process of training a neural network is equivalent to solving an optimization problem. It is often formulated as a minimization problem over $j = 1, 2, \ldots, m$ training examples, where the function we want to minimize is some loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) : \mathbb{R}^{d_L} \times \mathbb{R}^{d_L} \to \mathbb{R}$,

$$\hat{\theta} = \arg\min_{\theta} \sum_{j=1}^{m} \mathcal{L}(\mathbf{\Phi}(\mathbf{x}_j; \theta), \mathbf{y}_j). \tag{6.2.1}$$

The goal is that after training $\mathbf{\Phi}(\cdot, \hat{\theta})$ closely resembles $\mathbf{\Phi}^*$.

6.2.1 Regularisation

It can be advantageous to impose additional constraints on the parameters of the network. This is called regularisation and the regularised loss function canbe expressed as

$$\tilde{\mathcal{L}}(\hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \beta\Omega(\theta), \tag{6.2.2}$$

where $\Omega(\theta)$ is some penalty on the parameters θ measured in some norm and $\beta \in [0, \infty)$ is a hyperparameter called the regularisation factor or regularisation strength. The larger β, the more regularisation is imposed. Common norm penalties include the L^2 and L^1 norm. The L^2 regularisation drives the weights of the network closer to the origin and promotes networks with weights of similar magnitude. It can be imposed with $\Omega_{L^2}(\theta) = \frac{1}{2}\|\mathbf{w}_i\|_2^2$, where $\mathbf{w}_i$ are the weights of the i:th layer. This regularisation strategy is commonly known as ridge regression or Tikhonov regularisation. In comparison, the L^1 regularisation promotes a more sparse solution than the L^2 one. Similarly, the L^1 regularisation penalizes the weights of the network as $\Omega_{L^1}(\theta) = \|\mathbf{w}_i\|_1$ [11].

6.2.2 Optimization Algorithms

Due to the non-linear activation functions, the minimization problem in Eq. 6.2.1 often becomes non-convex. Unlike the solution to a convex optimization problem, which in theory always converges to a global minimum, the solution to a non-convex optimization problem is not guaranteed to converge to a global minimum. Thus, iterative and often only first order optimization algorithms are used to train neural networks. The latter due to higher order methods being too computationally expensive.

The first step in training a neural network is to propagate the input $\mathbf{x}$ forward in the network to produce $\hat{\mathbf{y}}$ yielding a scalar-valued loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$. The second step is to compute the gradient with respect to the network parameters $\nabla_\theta \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$, often referred to as back-propagation. These gradients give a direction in which the parameters should be changed to improve the computed loss. The last step is to use the gradients in an optimization algorithm that is responsible for updating the parameters according to some scheme. This procedure is then repeated for each training example. Once this done, we say that the algorithm has trained for one epoch.

We now present two optimization algorithms common in training neural networks and deep models, namely stochastic gradient descent (SGD) and Adam. The Adam optimization algorithm was used to train the networks considered in this thesis.

6.2.2.1 Stochastic Gradient Descent

This algorithm estimates the gradient by computing the loss for a randomly selected training sample $\mathbf{x}$ in each iteration. A variation of this optimizer is to instead choose m_s many random training samples, forming a randomly selected mini-batch $\{\mathbf{x}_1, \ldots, \mathbf{x}_{m_s}\}$ with corresponding targets $\{\mathbf{y}_1, \ldots, \mathbf{y}_{m_s}\}$. The gradient is then computed as the average gradient of the mini-batch. The parameters are updated by going in the negative gradient direction with the step size l from the current iterate i. This step size l is called the learning rate, which for SGD can be either fixed or changed with the iteration. The gradient estimation and parameter update are thus calculated as

$$\begin{aligned} \mathbf{g} &\leftarrow \frac{1}{m_s} \nabla_\theta \left(\sum_{j=1}^{m_s} \mathcal{L}(\mathbf{\Phi}(\mathbf{x}_j; \theta), \mathbf{y}_j) \right) \\ \theta_{i+1} &\leftarrow \theta_i - l\mathbf{g}, \end{aligned} \tag{6.2.3}$$

which then is iterated with a new mini-batch each iteration until some stopping criterion is met [11].

6.2.2.2 Adam

The Adam optimizer is a combination of the previously popular methods AdaGrad [10] and RMSProp [14] and is one of the most common methods employed in deep learning frameworks. In comparison to SGD, Adam is an adaptive learning rate optimization algorithm. The Adam optimizer is identical to SGD when it comes to selecting a mini-batch and estimating the gradients, i.e. as the top equation in Eq. 6.2.3. It then uses the method of momentum, which accumulates an exponentially decaying moving average of past gradients. More specifically, it computes the first- and second-order momentum of the gradients as well as a bias correction term for each of them. The moment estimates, correction

terms and parameters are updates for each iterate i and time step t as

$$\begin{aligned} t &\leftarrow t+1 \\ \mathbf{s} &\leftarrow \rho_1 \mathbf{s} + (1-\rho_1)\mathbf{g} \\ \mathbf{r} &\leftarrow \rho_2 \mathbf{r} + (1-\rho_2)\mathbf{g} \odot \mathbf{g} \\ \hat{\mathbf{s}} &\leftarrow \frac{\mathbf{s}}{1-\rho_1^t} \\ \hat{\mathbf{r}} &\leftarrow \frac{\mathbf{r}}{1-\rho_2^t} \\ \Delta\theta &\leftarrow -l\frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta} \\ \theta_{i+1} &\leftarrow \theta_i + \Delta\theta \end{aligned} \tag{6.2.4}$$

where $\odot$ denotes the elementwise vector multiplication, $\mathbf{s}$ and $\mathbf{r}$ the biased first- and second-order moment estimates with corresponding correction terms $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$. ρ_1 and ρ_2 denote the exponential decay rates for the moment estimates and δ is a constant used for numerical stability. The method is initialized with $\mathbf{s} = \mathbf{r} = \mathbf{0}$ and $t = 0$ [18, 11].

6.3 Learning Koopman Invariant Subspaces

This section describes how neural networks can be used for Koopman spectral analysis. The neural networks described in this section were implemented in Python 3.5.6 using the package TensorFlow 2.1.0 [1].

To find suitable observable functions for a given non-linear problem is an open challenge. One potential method is to learn the functions from data. More specifically, we are interested in learning parameters of a neural network such that it resembles an appropriate vector-valued observable function. This data-driven approach requires first and foremost enough useful data. With useful data we mean that it has to be of a certain quality, e.g. not include too much measurement errors, and also that it is connected to the problem we want to solve. Gathering this data can in itself be a difficult and tedious process. In some cases it can be expensive and technically demanding, and in other cases one might have vast amounts of data but it lacks in quality or requires a lot of preprocessing. However, since we are working with computer models of real world systems we can generate as much and as accurate data as needed from simulations.

Once the data has been synthesized, we want to learn the parameters of a neural network, but first a network structure has to be established. This means determining which activation functions to use, how many hidden layers it should consist of, and their dimensions. Building a neural network is not trivial as it is hard to know what structure is suitable for a particular problem. Typically, one creates and trains several networks with different structures and evaluates their performance by measuring how well they generalize to unseen data. Another factor that often is taken into account when choosing a network structure is the complexity of the network, i.e. how many parameters it has. Often, there exists a trade-off between how well the network generalizes to unseen data and how complex the network is.

The next step is to determine which loss function to use. Typically, these are also problem-dependent, but again, there exist loss functions that have been proven to work well for certain problems. The loss function we used will be presented in more detail in the next section and will be referred to as the Koopman loss.

6.3.1 Koopman Loss

The overall idea is to map the input data $\mathbf{x}_k$ to $\mathbf{g}(\mathbf{x}_k)$, where $\mathbf{g} : \mathbb{R}^{n_x} \to \mathbb{R}^n$ is a neural network and n is a hyperparameter. With DMD, $\mathbf{g}(\mathbf{x}_k)$ is evolved one time step to $\mathbf{g}(\mathbf{x}_{k+1})$ according to Eq. (5.1.22). There is no easy way to go back from $\mathbf{g}(\mathbf{x}_{k+1})$ to our original states $\mathbf{x}_{k+1}$, since $\mathbf{g}$ is a neural network. Therefore a second neural network $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^{n_x}$ is introduced to represent the inverse mapping from observables back to the states such that $\mathbf{x}_k = \mathbf{h}(\mathbf{g}(\mathbf{x}_k))$. This is illustrated in Fig. 6.1. We will refer to $\mathbf{g}$ as the encoder and to $\mathbf{h}$ as the decoder.

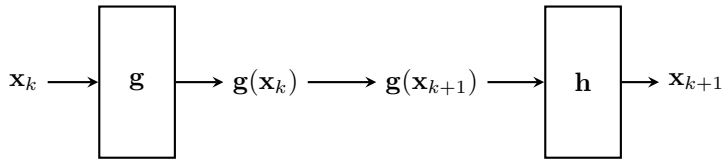


Figure 6.1: *Illustration of how the neural networks* $\mathbf{g}$ *and* $\mathbf{h}$ *are used for prediction with the Koopman method. The boxes represent neural networks.*

Consider the data matrices X, Y defined by Eq. (5.1.17). The loss function of $\mathbf{g}$, $\mathcal{L}_\mathbf{g}$, should train the network to span a Koopman invariant subspace. From Thm. 5.1.1 this is equivalent to the existence of a matrix $G \in \mathbb{R}^{n\times n}$, such that $G\mathbf{g}(\mathbf{x}_k) = \mathbf{g}(\mathbf{f}(\mathbf{x}_k)) = \mathbf{g}(\mathbf{x}_{k+1})$. From Thm. 5.1.2 we know that, under some mild conditions, if $m \to \infty$ then $\hat{A} = YX^\dagger \to G$. Therefore, if $\mathbf{g}$ spans a Koopman invariant subspace and m is sufficiently large, it should hold that $\hat{A}\mathbf{g}(\mathbf{x}_k) = \mathbf{g}(\mathbf{x}_{k+1})$. This results in the following loss function

$$\mathcal{L}_\mathbf{g} = \frac{1}{n}\|\mathbf{g}(\mathbf{x}_{k+1}) - \hat{A}\mathbf{g}(\mathbf{x}_k)\|_2^2, \tag{6.3.1}$$

which promotes a neural network representation of $\mathbf{g}$ that spans a Koopman invariant subspace. This loss function can be seen as finding $\mathbf{g}$ such that $\mathbf{g}(\mathbf{x}_k)$ evolves linearly in time. Obviously it is desired that this holds for many time steps. To promote that, Eq. (6.3.1) is modified into what will be referred to as the linearity loss, namely

$$\mathcal{L}_{lin} = \frac{1}{m}\sum_{k=1}^{m}\frac{1}{n}\|\mathbf{g}(\mathbf{x}_{k+1}) - \hat{A}^k\mathbf{g}(\mathbf{x}_1)\|_2^2. \tag{6.3.2}$$

For the decoder network, $\mathbf{h}$, we want to train it such that it reconstructs the original states. A suitable reconstruction loss is simply

$$\mathcal{L}_{rec} = \frac{1}{n_x}\|\mathbf{x}_k - \mathbf{h}(\mathbf{g}(\mathbf{x}_k))\|_2^2, \tag{6.3.3}$$

which is small if $\mathbf{h}$ is able to correctly reconstruct $\mathbf{x}_k$ from $\mathbf{g}(\mathbf{x}_k)$.

In addition to the linearity loss and the reconstruction loss, a prediction loss is also used. This can be stated as

$$\mathcal{L}_{pred} = \frac{1}{n_x}\|\mathbf{x}_{k+1} - \mathbf{h}(\hat{A}\mathbf{g}(\mathbf{x}_k))\|_2^2. \tag{6.3.4}$$

Analogously to the linearity loss the prediction loss is rewritten to promote multi-step predictions, which gives

$$\mathcal{L}_{pred} = \frac{1}{m}\sum_{k=1}^{m_{pred}}\frac{1}{n}\|\mathbf{x}_{k+1} - \mathbf{h}(\hat{A}^k\mathbf{g}(\mathbf{x}_1))\|_2^2, \tag{6.3.5}$$

where $m_{pred} \leq m$ will be tuned as a hyperparameter.

Adding these losses together, possibly with individual scaling factors $\gamma \in \mathbb{R}$, the final loss function can be stated as

$$\mathcal{L} = \gamma_{lin}\mathcal{L}_{lin} + \gamma_{rec}\mathcal{L}_{rec} + \gamma_{pred}\mathcal{L}_{pred}. \tag{6.3.6}$$

However, throughout this thesis we will consider these scaling factors to be equal to one.

This loss function was suggested by [20]. In that paper it is also suggested to pre-train the neural networks with the reconstruction loss, a suggestion that will be used throughout this thesis.

6.3.2 Network Structure

Throughout this report the same network structure has been used (unless stated otherwise). Before the general network structure is presented, some restrictions that were made have to be stated. First and foremost, to reduce the number of hyperparameters identical network structures were used for both the encoder and the decoder network. Only affine layers were used, except for the activation functions of course. The networks are also restricted to one hidden layer with the same input and output dimension. This gives a network structure that can be described as a function $\mathbf{\Phi} : \mathbb{R}^{n_x} \to \mathbb{R}^n$,

$$\mathbf{\Phi}(\mathbf{x}) = \phi_{out} \circ \phi_{ReLU} \circ \phi_{hidden} \circ \phi_{ReLU} \circ \phi_{in}(\mathbf{x}), \tag{6.3.7}$$

where ϕ_{ReLU} denotes the activation function for which ReLU has been used, i.e. $\phi_{ReLU}(\mathbf{x}) = \max(\mathbf{0}, \mathbf{x})$, where the max operation is performed elementwise such that $\phi_{ReLU} : \mathbb{R}^{d_{hidden}} \to \mathbb{R}^{d_{hidden}}$. For the affine layers it holds that $\phi_{in} : \mathbb{R}^{n_x} \to \mathbb{R}^{d_{hidden}}$, $\phi_{hidden} : \mathbb{R}^{d_{hidden}} \to \mathbb{R}^{d_{hidden}}$ and $\phi_{out} : \mathbb{R}^{d_{hidden}} \to \mathbb{R}^n$. The affine layers can be written as

$$\phi(\mathbf{x}; \mathbf{w}, \mathbf{b}) = \mathbf{x}^T\mathbf{w} + \mathbf{b}, \tag{6.3.8}$$

with $\mathbf{w}_{in} \in \mathbb{R}^{n_x \times d_{hidden}}$, $\mathbf{w}_{hidden} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$, $\mathbf{w}_{out} \in \mathbb{R}^{d_{hidden} \times n}$, $\mathbf{b}_{in} \in \mathbb{R}^{d_{hidden}}$, $\mathbf{b}_{hidden} \in \mathbb{R}^{d_{hidden}}$ and $\mathbf{b}_{out} \in \mathbb{R}^n$. The weights were initialized from a uniform distribution $\mathcal{U}(-\sqrt{1/d_{in}}, \sqrt{1/d_{in}})$, where d_{in} is the input dimension to each specific layer, e.g. for $\mathbf{w}_{in}$ $d_{in} = n_x$ and for $\mathbf{w}_{hidden}$ $d_{in} = d_{hidden}$.

6.4 Numerical Examples

We now present a few examples that show the capabilities of the Koopman method when the vector-valued observable functions are learnt from data. We also highlight similarities and differences compared to DMD.

Example 6.4.1 (Basic neural network as observable function)**.** Consider the damped dual mass system in Fig. 2.1 with a sine-shaped input force of amplitude 10 N and frequency 0.1 Hz. The initial condition and system constants are the same as in Sect. 4.3. Note that a different input force with lower frequency has been used for this example compared to most examples in Sect. 4.3. Suppose that the states are chosen as the positions and velocities of both masses. We know that there exists an analytical solution to this system and that there is a linear relation between these states from one point in time to the next. We also know that the DMD algorithm is capable of finding this linear relation. We will now show that the Koopman method with a basic neural network as the vector-valued observable function also is capable of predicting this system.

Let $\mathbf{g}$ be represented by a very simple neural network, namely a network without activation functions and only one affine layer without bias, i.e. $\mathbf{g}(\mathbf{x}) = \mathbf{x}^T\mathbf{w}_g$, where $\mathbf{w}_g \in \mathbb{R}^{n_x \times n_x}$. The elements of $\mathbf{w_g}$ are initialized from a uniform distribution $\mathcal{U}(-\sqrt{1/n_x}, \sqrt{1/n_x})$. Define $\mathbf{h}$ analogously to $\mathbf{g}$. Even though we know that the DMD algorithm can evolve this system without any observable functions, which would be equivalent to $\mathbf{w}_g = \mathbf{w}_h = I$, there is no guarantee that the optimization algorithm of choice (Adam) will find that solution. It turns out that Adam indeed does not find that specific solution.

The networks were trained first with just the reconstruction loss, Eq. (6.3.3), for 160 epochs with learning rate 10^{-2} and then using the Koopman loss, Eq. (6.3.6), for 2000 epochs with learning rate $2.5 \cdot 10^{-4}$. Adam was used in both cases. Data from the first 10 s were used as a training set and the remaining 10 s where used for validation. The system was sampled with a frequency of 50 Hz and $m = 500$. The predictions of the position of the second mass made with the trained networks can be found in Fig. 6.2.

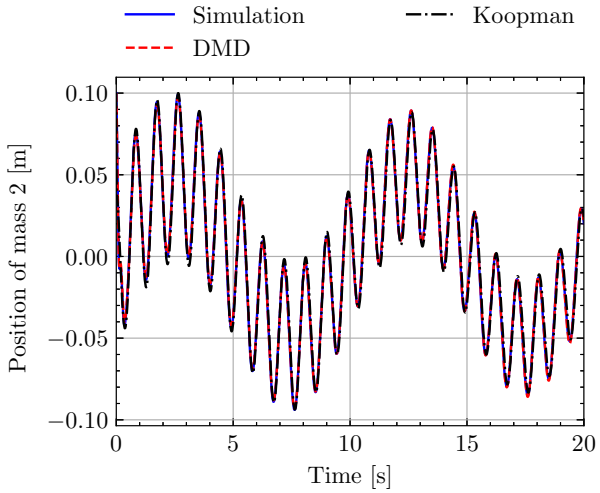


Figure 6.2: *Predictions of the position of the second mass over time done with DMD and the Koopman method, where a neural network was used to identify the observable functions. For comparison the simulated position is also presented. Data from the first 10 s was used for training and the remaining 10 s for validation.*

The predictions made with the Koopman method closely resemble the simulated solution, but not as closely as the predictions made with DMD. In Tab. 6.1 the prediction errors are presented. It is clear that both methods come close to the simulated solution, but the prediction error received when using DMD is roughly one order of magnitude smaller than when the Koopman method was used.

Table 6.1: *Prediction errors for the Koopman method, where a neural network has been used to identify observables, compared to DMD.*

	Training		Validation	
	DMD	Koopman	DMD	Koopman
$\frac{1}{N}\|\hat{x}_1 - x_1^{sim}\|_2$	$5.3 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$2.3 \cdot 10^{-5}$	$1.7 \cdot 10^{-4}$
$\frac{1}{N}\|\hat{x}_2 - x_2^{sim}\|_2$	$5.9 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$	$2.6 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$

In conclusion, to use a neural networks to find observable functions for a linear system does not make sense. The method is simply too complicated and there are better options. This does however show that a neural network is able to find observables such that there is a linear relation between the states at different points in time.

Example 6.4.2 (Learn known observable functions)**.** We now study how a neural network can be used to find observable functions for the dynamical system presented in Ex. 5.3.3, which is non-linear in x_2. The initial value and parameters from Ex. 5.3.4 was used. The system was sampled with a

frequency of 50 Hz for 12 s. The first 6 s of the data was discarded due to x_2 being almost constant. The remaining 6 s of data was scaled with a factor 10^2 for numerical reasons. Note that the predictions were scaled back afterwards.

The network structure from Sect. 6.3.2 was used with $w_{hidden} = 80$ and $n = 6$. The networks were trained first with just the reconstruction loss, Eq. (6.3.3), for 1000 epochs with learning rate $8 \cdot 10^{-5}$ and then using the Koopman loss, Eq. (6.3.6), for 1000 epochs with learning rate $2 \cdot 10^{-6}$. Adam was used in both cases.

Fig. 6.3 shows the predictions made with the Koopman method, where it can be compared to predictions made with DMD as well as the analytical solution.

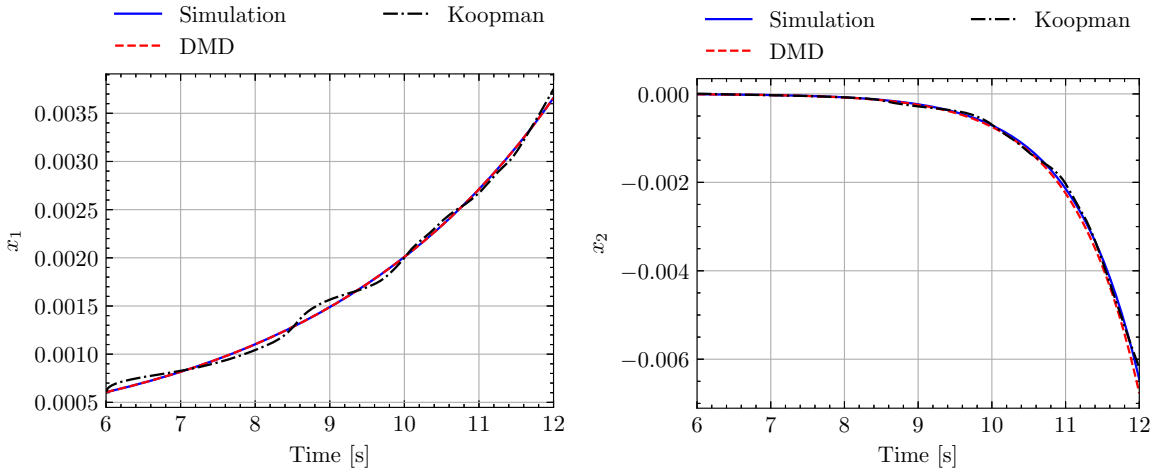


Figure 6.3: *Predictions of the states x_1 (6.3a) and x_2 (6.3b).*

Predictions made with the Koopman method follow the general trend of the solution, but there seem to be a ringing behaviour around the trend line. For x_1, which can be evolved linearly in time, DMD finds the correct trajectory as expected, but for x_2 it deviates more. From Ex. 5.3.3 we know that there exists an analytical linear solution to this system if the observable functions from Eq. (5.3.2) are chosen. The neural network has clearly not found that exact solution, but it still manages to find observables that make it possible to follow a general trend line. The prediction errors are presented in Tab. 6.2. For x_2 the Koopman method comes closer to the real solution than DMD, but for x_1 DMD yields a much lower prediction error. It is worth noting that prediction errors from the Koopman method are similar for both x_1 and x_2.

Table 6.2: *Prediction errors for the Koopman method, where a neural network has been used to identify observables, compared to DMD.*

	DMD	Koopman
$\frac{1}{N}\|\hat{x}_1 - x_1^{an}\|_2$	$7.3 \cdot 10^{-18}$	$2.7 \cdot 10^{-6}$
$\frac{1}{N}\|\hat{x}_2 - x_2^{an}\|_2$	$5.0 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$

Example 6.4.3 (Learning unknown observable functions for a non-linear system)**.** We now consider the non-linear damped dual mass system from Ex. 4.3.7. The system was sampled with a frequency of 100 Hz for 40 s, where the first 20 s were used for training and the remaining 20 s for validation.

The network structure from Sect. 6.3.2 was used with $w_{hidden} = 60$ and $n = 21$. The networks were trained first with just the reconstruction loss, Eq. (6.3.3), for 300 epochs with learning rate $5 \cdot 10^{-3}$ and then using the Koopman loss, Eq. (6.3.6), for 400 epochs with learning rate $8 \cdot 10^{-5}$. Adam was used in both cases.

Fig. 6.4 shows the predictions of the position of the first mass made with the Koopman method, where it can be compared to predictions made with DMD as well as the simulated position over time.

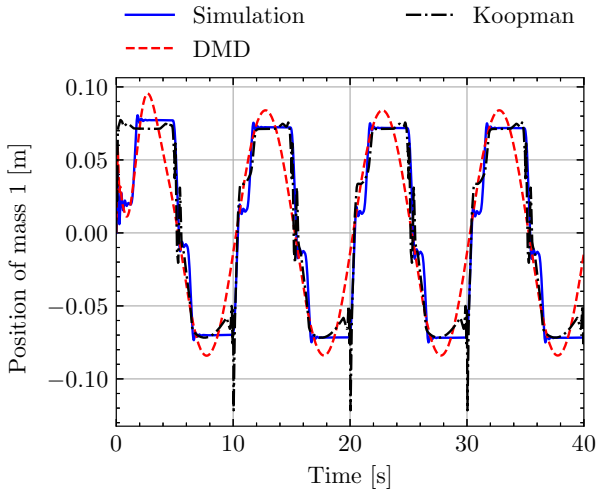


Figure 6.4: *Predictions of the position of the first mass over time done with DMD and the Koopman method, where a neural network was used to identify the observable functions. For comparison the simulated position is also presented. Data from the first 20 s was used for training and the remaining 20 s for validation.*

The predicted position over time given by the Koopman method is able to follow the simulated position of the first mass much more accurately than the prediction from DMD. The DMD method produces a prediction that looks like a simple sine-wave that follows the general amplitude and frequency, whereas the Koopman method manages to reconstruct some of the non-linearities and follow the trajectory of the position more closely. This is made very clear in Tab. 6.3 where the prediction errors are presented. The prediction errors from the Koopman method is almost a factor two smaller when compared to the DMD method.

Regarding the spikes every 10:th second, one might think that this is a result of the training set ending at 20 s where the mass goes from a stationary to a moving state. However, additional tests revealed that this was not the case as the spikes were still present if the training set was made smaller (up to 19 s).

Table 6.3: *Prediction errors for the Koopman method, where a neural network has been used to identify observables, compared to DMD.*

	Training		Validation	
	DMD	Koopman	DMD	Koopman
$\frac{1}{N}\|\hat{x}_1 - x_1^{an}\|_2$	$4.9 \cdot 10^{-4}$	$4.1 \cdot 10^{-4}$	$5.0 \cdot 10^{-4}$	$3.0 \cdot 10^{-4}$
$\frac{1}{N}\|\hat{x}_2 - x_2^{an}\|_2$	$6.1 \cdot 10^{-4}$	$5.1 \cdot 10^{-4}$	$6.2 \cdot 10^{-4}$	$3.6 \cdot 10^{-4}$

Chapter 7

Data Sets

The 13 data sets presented in this chapter can be divided into nine training, two validation and two test sets. We refer to the training sets as the ones used in the construction of the data matrices for the DMD and Koopman method. The validation set is used to tune potential hyperparameters. The test sets consists of data not previously seen by the fitted models and are used to evaluate how well the methods generalize to new data. Each data set was co-simulated in PyFMI 2.4.0 using Dymola's modified DASSL solver with a relative tolerance of 10^{-6}. The solution was then sampled with a frequency of 1000 Hz, which results in an equidistant grid with a step size of 10^{-3}. The three FMUs used in each co-simulation was one input generator FMU, which was different for each data set, one software controller FMU for the FABV and one FMU of the FABV itself. The input FMUs were constructed by first creating a Modelica model in OpenModelica version 1.14.1 and then exported to FMUs using Dymola 2019. The C compiler used in both tools was GCC 9.3.0.

7.1 Setup

In order to perform the DMD and Koopman method a large amount of input and output data pairs from the FABV needs to be generated. The input signals to the FABV are, as described in Sect. 2.2, two set point currents I_1 and I_2, and the output is a pressure p. The task is to create a sequence of set point currents and record the output pressure of the FABV. However, creating a valid sequence of inputs to the FABV is not feasible since they originate from a controller in a feedback loop, see Fig. 7.1. The solution to this problem is to instead create a sequence of input signals to the software controller and then extract the control signals I_1 and I_2 from that simulation. More specifically, the input signal to the software controller is an interpretable pressure set point p_{sp}, which is generated from an input generator FMU.

The input FMUs were constructed from suitable Modelica components in OpenModelica and then exported to co-simulation FMUs using Dymola. These FMUs were exported using Dymola's modified DASSL solver. The reason as to why Dymola was used was due to the fact that OpenModelica does not support the latest FMI version (2.0), which is used by the software controller and FABV FMUs. It is necessary for the input FMUs to support FMI 2.0 because all FMUs in a co-simulation must have the same FMI version. The FMUs of the controller and the FABV were supplied by Haldex.

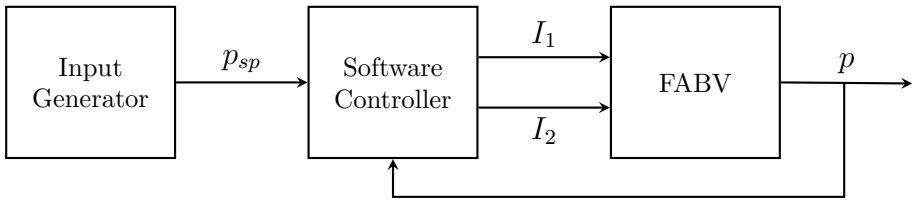
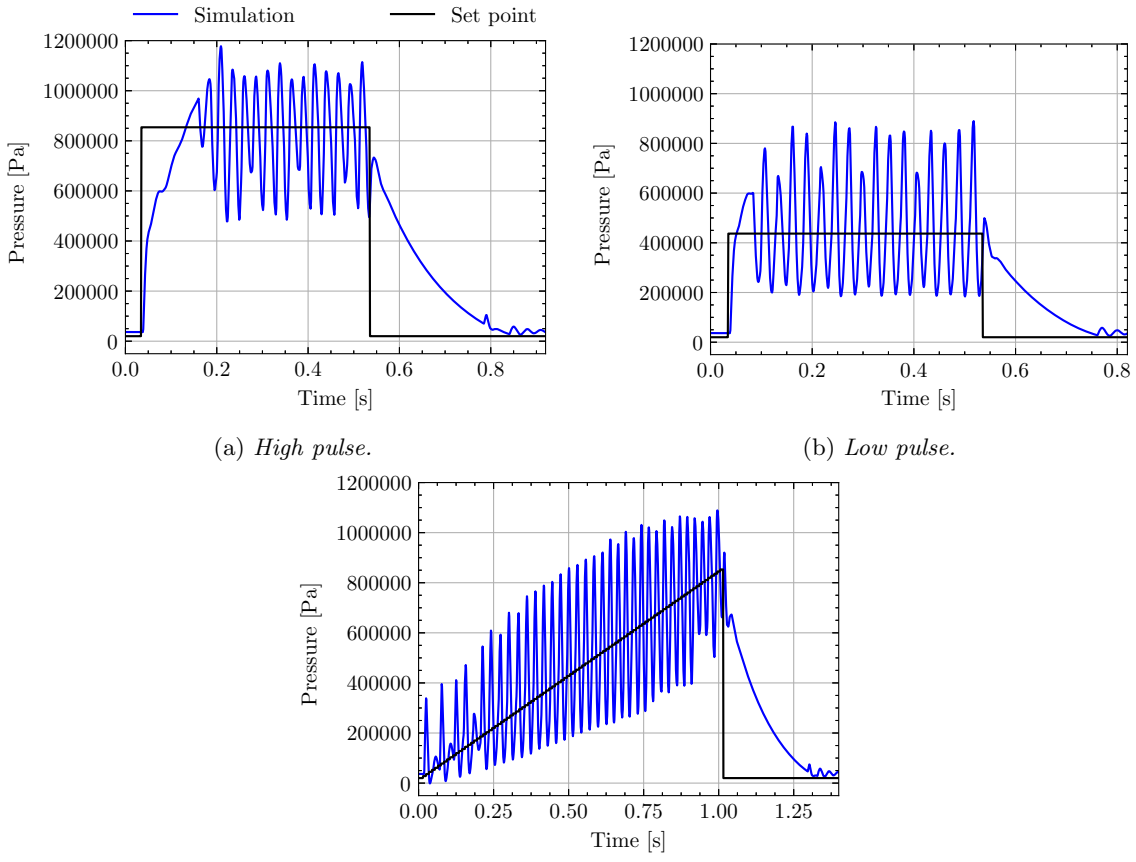


Figure 7.1: *Sketch of the relevant inputs and outputs to and from the FMUs as well as the connections between them. Each box represents an FMU. The input generator FMU outputs a pressure set point p_{sp} that is fed to the software controller, which in turn regulates the output pressure p of the FABV using the set point currents I_1 and I_2.*

7.2 Training Data

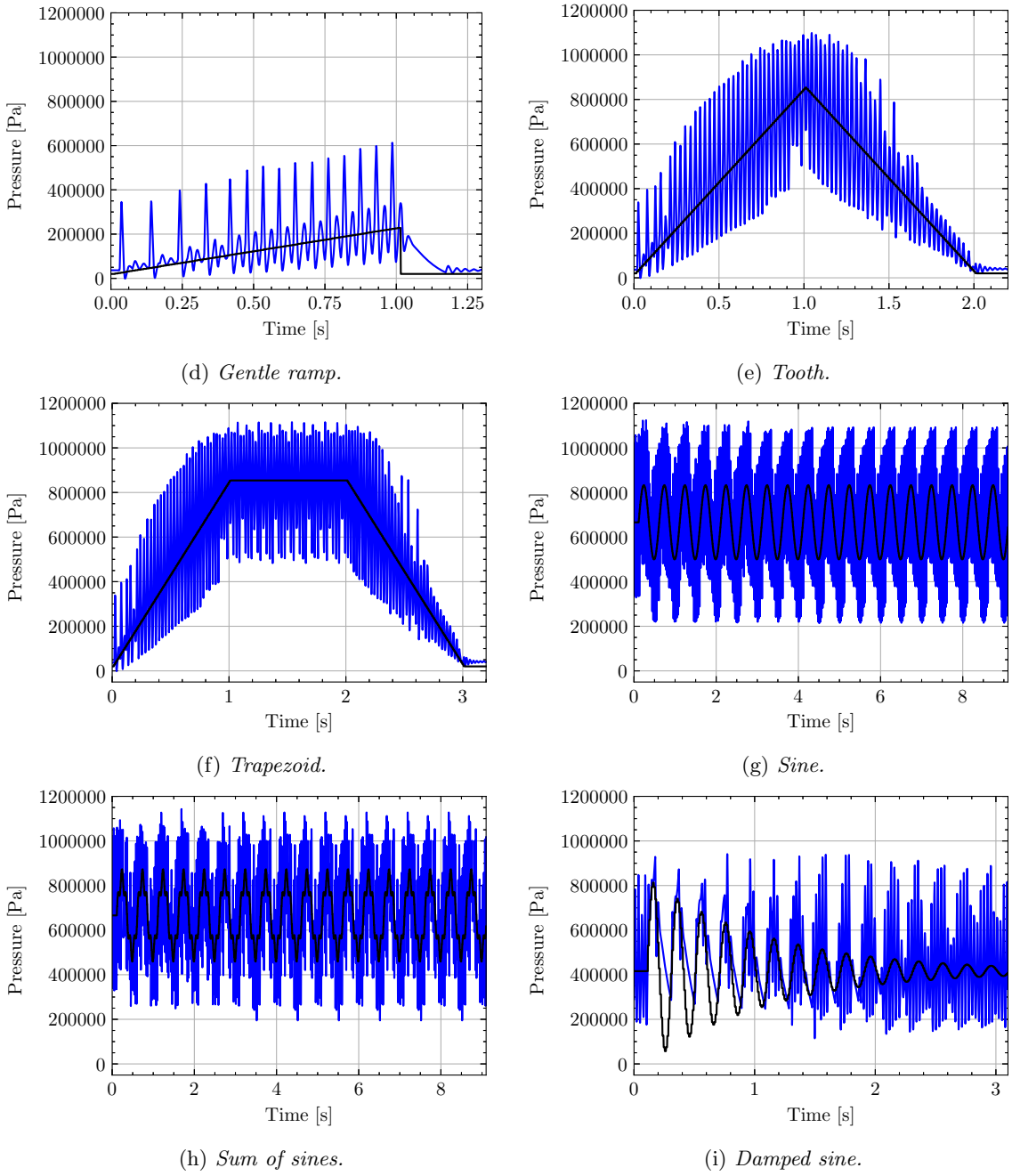
In order to make the models perform well on unseen data, a wide variety of different input signals were used. The pressure set points considered in the training set thus consist of a variety of different pulses, ramps, trapezoids and sine waves. The pressure set point and the simulated pressure for each input signal is presented in Fig. 7.2. Evidently, the the FABV model is highly non-linear. The general trend is that the output pressure oscillates with a relatively large magnitude and high frequency around the set point.



(a) *High pulse.*

(b) *Low pulse.*

(c) *Steep ramp.*



(d) *Gentle ramp.*

(e) *Tooth.*

(f) *Trapezoid.*

(g) *Sine.*

(h) *Sum of sines.*

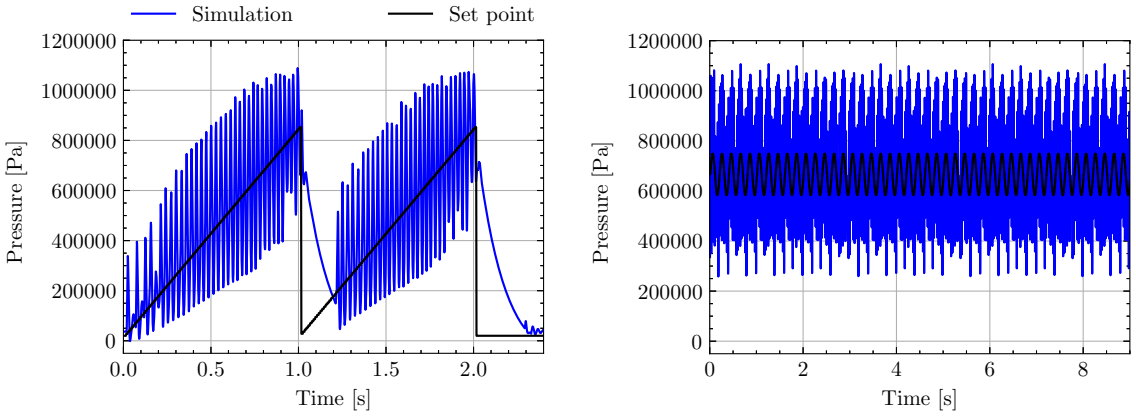
(i) *Damped sine.*

Figure 7.2: *Sequences of pressure set points together with the simulated pressure output by the FABV used in the training.*

Note that the pressure set points shown in Fig. 7.2 are not the input signals used for the DMD and Koopman method, since they are inputs to the software controller and not the FABV. We are however able to extract the correct inputs I_1 and I_2 from these co-simulations.

7.3 Validation Data

The validation set was generated by feeding a saw tooth and a sine signal to the software controller. The set point and simulated pressures are shown in Fig. 7.3. The purpose of the validation data set is to tune the hyperparameters of the different methods.

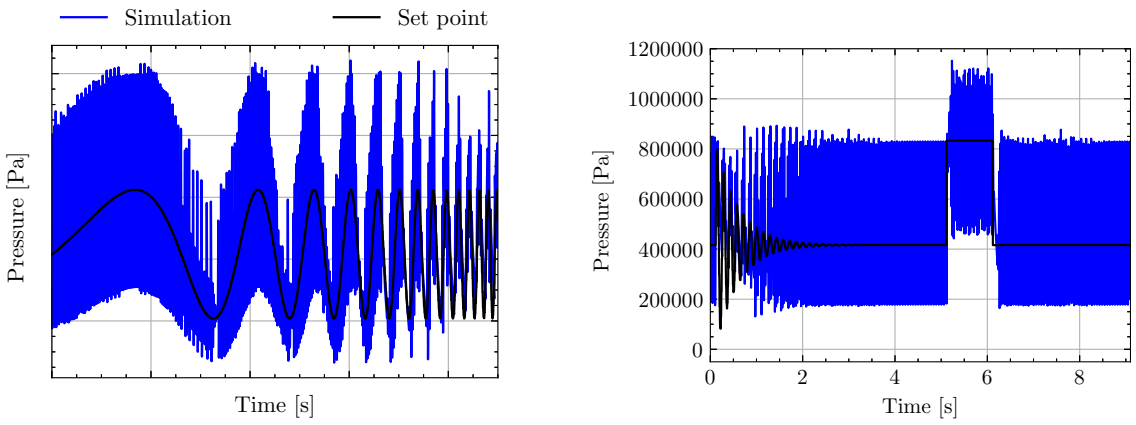


(a) *Saw tooth.* (b) *Sine.*

Figure 7.3: *Pressure set point and simulated pressure for the validation data set.*

7.4 Test Data

The purpose of the test sets are to evaluate how well the methods generalize to unseen data. The first test set seen in Fig. 7.4a has a pressure set point consisting of a sine wave with constant amplitude and increasing frequency. This signal will be referred to as the chirp signal. The reason why the chirp signal was used as a test set is due to it being a signal that is frequently used by Haldex to benchmark different iterations of their FABV model. Due to this data being confidential, the axes on all figures including the chirp signal has been removed. The second test set in Fig. 7.4b starts of with a pressure set point described by a damped sine wave, and once it has been completely damped, we request a pressure described by a pulse.

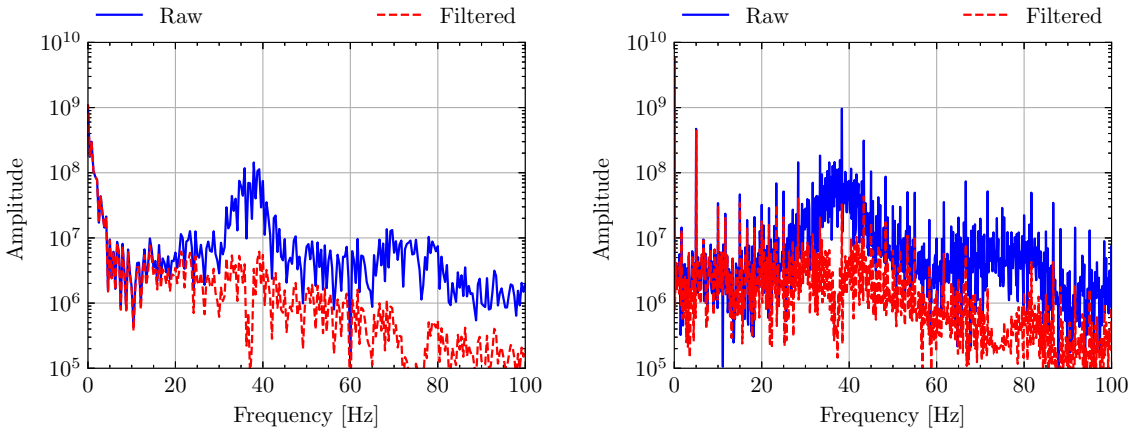


(a) *Chirp.* (b) *Damped sine and pulse.*

Figure 7.4: *Pressure set point and simulated pressure for the two data sets used to test the methods.*

7.5 Frequency Spectrum

In all data sets, regardless of input, the pressure signal contains a high amplitude high frequency component, even when the set point value is constant. This is not necessarily a problem, since the pressure signal is filtered in the software controller. However, when our models are evaluated we are mainly interested in how well they follow the general trend. It is of lesser interest how well they fit to the higher frequencies. To better understand the frequency spectrum of the pressure signal the Fast Fourier Transform (FFT) was used and the result from two different input signals (saw tooth and sine) are presented as the blue solid lines in Fig. 7.5.



(a) *Saw tooth.* (b) *Sine.*

Figure 7.5: *Frequency spectrum of the pressure signal for the data in the validation set. The blue solid line shows the FFT of the raw data and the red dashed line of the filtered data.*

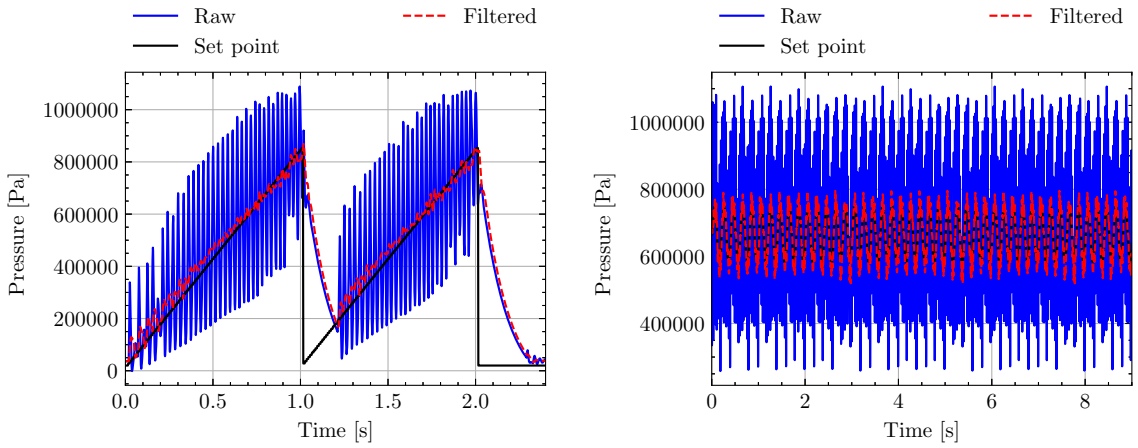
Both pressure signals contain a strong component with a frequency of around 37 Hz. This aligns well with what can be found in Fig. 7.3. For both inputs, the pressure contains a high amplitude

component with a period of around 27 time steps. With a step size of 10^{-3} this yields a period of approximately $27 \cdot 10^{-3} = 0.027$ s, which is equivalent to a frequency of 37 Hz. One way to remove this component is to use a simple moving average filter, which for each step computes the mean over the last 27 steps, more specifically

$$p_t = \frac{1}{27} \sum_{i=t-26}^{t} p_i. \tag{7.5.1}$$

Note that the first 26 data points in the filtered data set has to be discarded due to them not being valid. It is also worth mentioning some of the drawbacks of such a moving average filter. Primarily it comes down to the introduced delay. If the sum is not centred around the current point in time, a delay equal to half of the window size will be introduced, in this case $26/2 = 13$ time steps or 0.013 s. When our models are evaluated the delay will not be an issue. This is due to the fact that we only compare filtered data to filtered data, i.e. the delay will be present on both sides. A centred moving average filter can not be used since it requires future states to be known, which is not the case in an actual simulation. For instance, a centred moving average filter of width 27 requires the 13 future states to be known.

The result from the FFT of the filtered signals can be found in Fig. 7.5 as the red dashed lines. The filtered distributions closely follow their unfiltered counter parts up until around 25 Hz, where the amplitude of those higher frequency components start to get dampened. Frequencies close to 37 Hz are as expected heavily dampened. Frequencies above 37 Hz are also dampened, but they are also of low interest. Fig. 7.6 shows the unfiltered pressure signal together with the filtered signal.



(a) *Saw tooth.* (b) *Sine.*

Figure 7.6: *Simulated and filtered pressure signal for the validation set, where the blue solid line is unfiltered and dashed red line has been filtered with a running mean filter that takes the mean over the last 27 steps.*

From Fig. 7.6 it becomes clear that the simple moving average filter successfully removes the high frequency components. The overall trend still remains and is in fact much clearer. In conclusion, we will evaluate our models on both filtered and unfiltered data. This way we will be able to tell if the prediction errors come from lack in precision of the general trend or inaccurate reconstruction of the higher frequency components.

Chapter 8

Results

The results presented in this chapter were generated using implementations of the DMD and the Koopman method in Python 3.5.6 mainly using the packages NumPy 1.18.3, SciPy 1.4.1 and TensorFlow 2.1.0. More specifically, the computation of the DMD modes utilizes the SVD and eigendecomposition from NumPy and SciPy, respectively. Moreover, the version of TensorFlow used did not support differentiation of an eigendecomposition. However, an implementation of this based on [8] existed in the nightly build 2.2.0-dev20200419, from which we copied the relevant code and used it in our own scripts [27].

Measuring Errors

There are multiple ways to measure the accuracy of a prediction. Different ways highlight different aspects, and it can thus be beneficial to use several. In this chapter we consider two ways of measuring the errors of the methods, namely by the absolute and relative 2-norm denoted by ε_a and ε_r, respectively. Consider the pressure $\mathbf{p} \in \mathbb{R}^N$ simulated with high-precision and the pressure $\hat{\mathbf{p}} \in \mathbb{R}^N$ computed using the DMD or Koopman method, where N is the number of time steps. Then, ε_a and ε_r are defined as

$$\varepsilon_a = \frac{1}{N}\|\hat{\mathbf{p}} - \mathbf{p}\|_2 \tag{8.0.1}$$

and

$$\varepsilon_r = \frac{\|\hat{\mathbf{p}} - \mathbf{p}\|_2}{\|\mathbf{p}\|_2}, \tag{8.0.2}$$

respectively.

Measuring Computational Time

When the time consumption is measured there are a few different alternatives to consider. One option is central processing unit (CPU) time, which measures the time the CPU was processing instructions of a computer program. This time measure does not include for instance waiting time for input/output operations. Another option is elapsed real time or wall-clock time. This refers to the total time the program was active, i.e. time from start to finish. For this thesis wall-clock time will be used. Time consumption will be presented as an average of 10 runs, where the chirp signal has been used as input. All presented computing times were done on a laptop with an Intel Xeon E-2276M processor and 32 GB of RAM.

8.1 Dynamic Mode Decomposition Method

This section presents a comparison between a truncated DMD solution and a high-fidelity simulated solution. We train the DMD using the raw training data given in Ch. 7 and study how different amounts of truncation of the data affect the solutions. Once a suitable truncation level has been found, we present the absolute and relative errors on all data sets, both unfiltered and when the filter described in Sect. 7.5 has been applied to the solutions.

8.1.1 Truncation

In this section we study how the performance of the method varies for different low-rank approximations of the training data. This is of interest because, as discussed in Rem. 4.1.3, depending on the value of n_x and available computer hardware, it can become infeasible to compute and store $\hat{A}$. In the case of the FABV there are $n_x = 2376$ number of states. Even though this is not considered large, in the setting of this thesis, the truncation analysis is still carried out since it is still an interesting experiment.

Using the training data described in Ch. 7 we construct the training data matrices $X, Y \in \mathbb{R}^{n_x \times m}$ according to Eq. (4.1.3) where $n_x = 2376$ and $m = 31133$. We use Thm. 4.1.2 and Alg. 2, where the goal is to find a low-rank approximation of X such that it closely resembles X. In Fig. 8.1 the magnitude of the singular values of X are shown. Evidently, the magnitude of the singular values decreases rapidly with increasing index and plateaus around index 220. This suggests that X truncated to rank $n_r < 220$ would resemble X well.

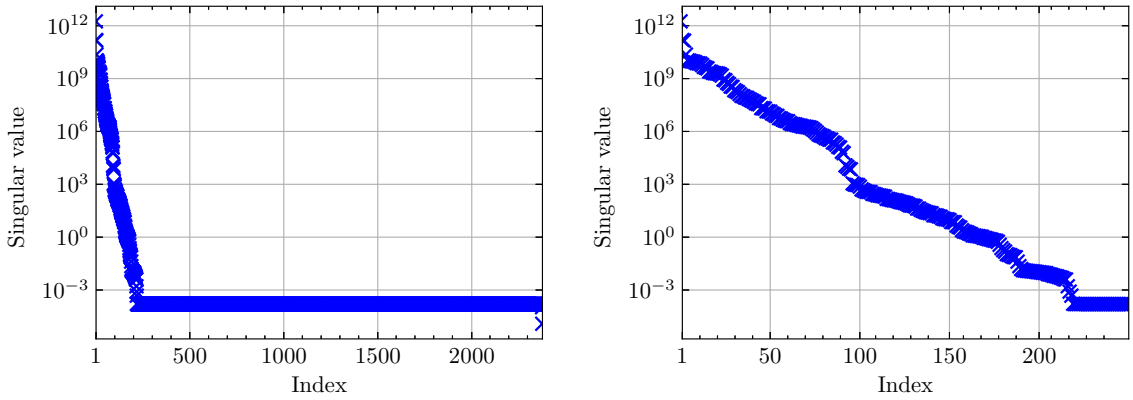


Figure 8.1: *(8.1a) Illustration of the magnitude of the singular values of the data matrix X and (8.1b) a zoomed in version.*

In order to determine the optimal n_r, i.e. the smallest one we may choose without losing too much information in X, we consider it a hyperparameter, which we tune using the training and validation data sets. This was done by computing the DMD using the training data for different values of n_r and study how the prediction error on the validation data set varied. The result from this study, where n_r was varied from 10 to 220 in steps of 10, is shown in Fig. 8.2-8.4. However, due to the error being infinite for $n_r > 180$, the figures only presents the errors for $n_r \leq 180$. The general trend of Fig. 8.3 is that the relative errors on both the training and validation data set decrease with increasing values of n_r up until $n_r = 50$, where they plateau. Studying the individual relative errors we see that one example stands out, namely the gentle ramp, with relatively large errors for small values of n_r. The error drops

significantly for all examples at $n_r = 50$. We therefore set $n_r = 50$. Using Thm. 4.1.3 we compute the relative error between X and its SVD truncated to rank n_r as $\sigma_{\nu+1}/\|X\|_2 = \sigma_{51}/\sigma_1 \approx 7.5 \cdot 10^{-6}$. This value is regarded as small, meaning that we are able to represents the original data matrix X well using only the first 50 left singular vectors, singular values and right singular vectors.

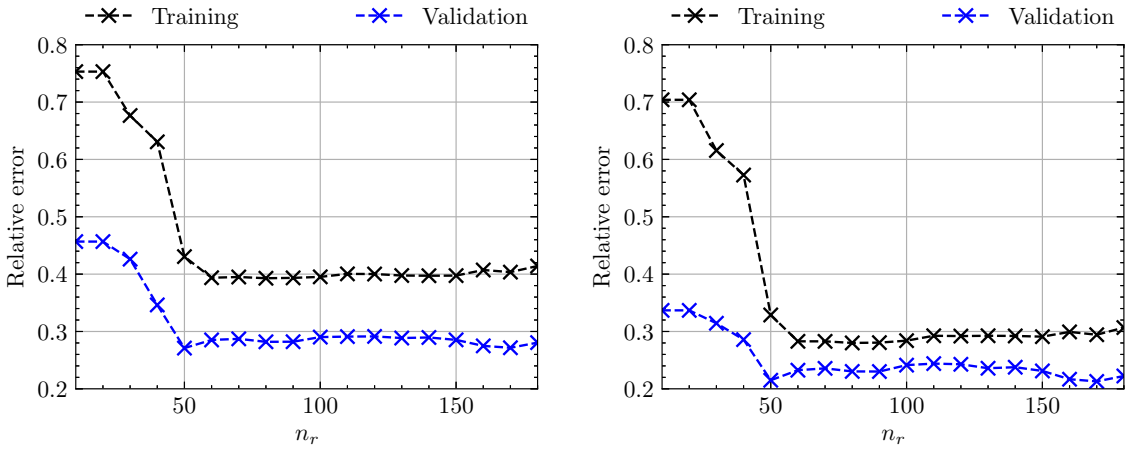


Figure 8.2: *Relative errors of training and validation data set computed as the mean of all examples in each set for different values of n_r. Fig. 8.2a shows when the results are unprocessed and Fig. 8.2b when they are.*

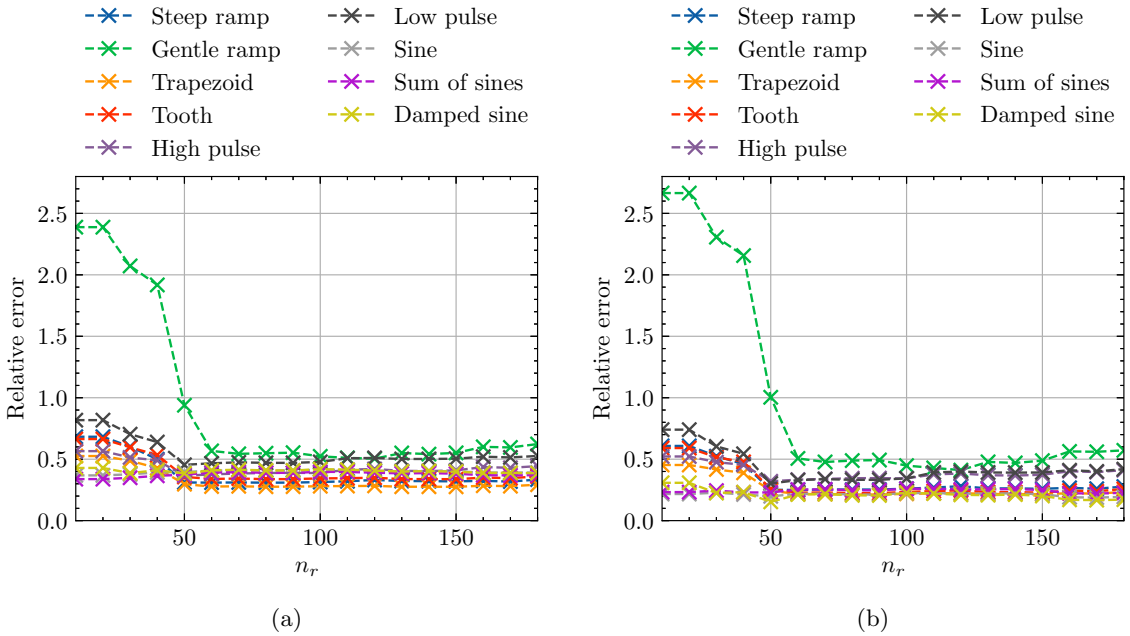


Figure 8.3: *Relative errors of each training examples using unprocessed results in 8.3a and processed in 8.3b*

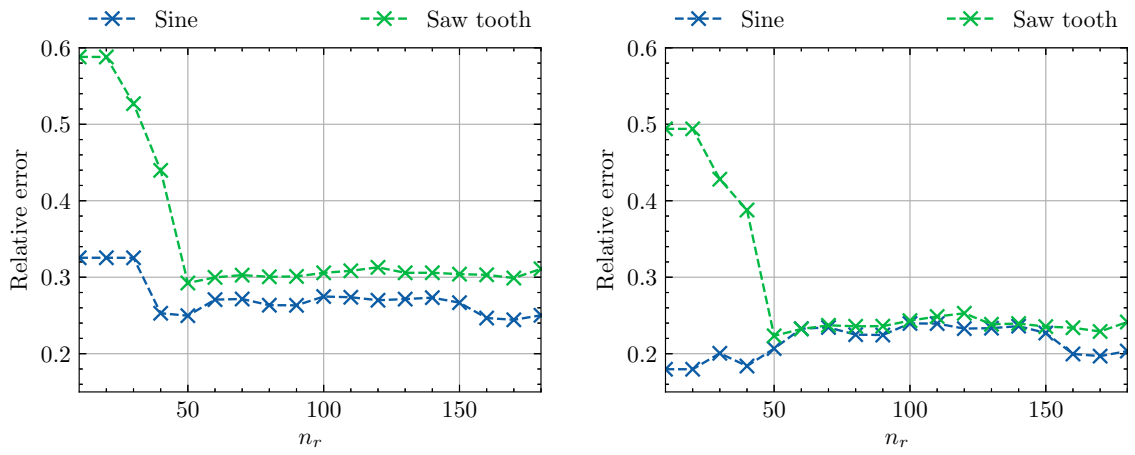


Figure 8.4: *Relative errors of each validation examples using unprocessed results in 8.4a and processed in 8.4b*

8.1.2 Temporal DMD Modes

The truncated DMD extracts n_r eigenvalues with corresponding DMD modes. The eigenvalues provide information about the exponential growth/decay as well as the oscillation of each mode. Using Eq. (4.1.12) we can calculate the frequency of each DMD mode. Since a sampling frequency of 1000 Hz was used, the highest possible frequency that can be reconstructed is $1000/2 = 500$ Hz. We therefore do not expect any modes to oscillate with a frequency higher than 500 Hz. Furthermore, an eigenvalue with an absolute value equal to one, which is equivalent to $\mu = 0$ in Eq. (4.1.12), indicates that the mode does not suffer from neither decaying nor growing effects in time.

Fig. 8.5 shows the frequencies of the extracted DMD modes and their corresponding exponential growth or decay. First and foremost, we see that each eigenvalue has an absolute value less than one. This means that all modes suffer from dampening effects, i.e. $\mu > 0$, which in turn is essential for the solution to not become infinitely for large times. Secondly, we note that there are five modes with an imaginary part equal to zero, which implies that they have a frequency of zero. These modes therefore purely constitute decaying dynamical characteristics that do not vary in time. We can see that the modes span the whole possible frequency spectrum and that they seem to be clustered around the lowest and highest frequencies. About half of the modes lie in the frequency span of 0 Hz to 100 Hz, and approximately a third lie around the maximum frequency 500 Hz. This implies that a significant part of the states of the FABV can be reconstructed using a linear combination of components with a frequency of less than 100 Hz, and that another large portion require components with frequency of ≥ 500 Hz.

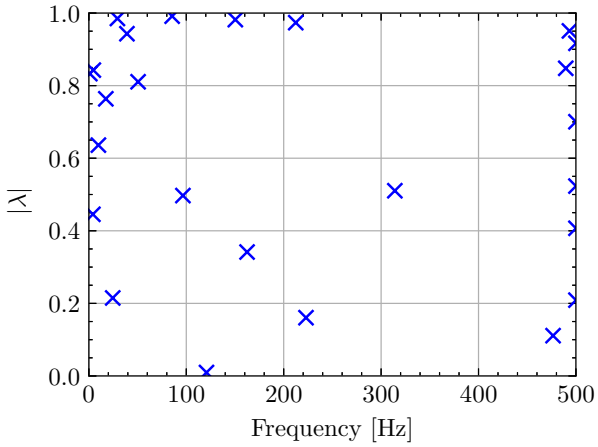
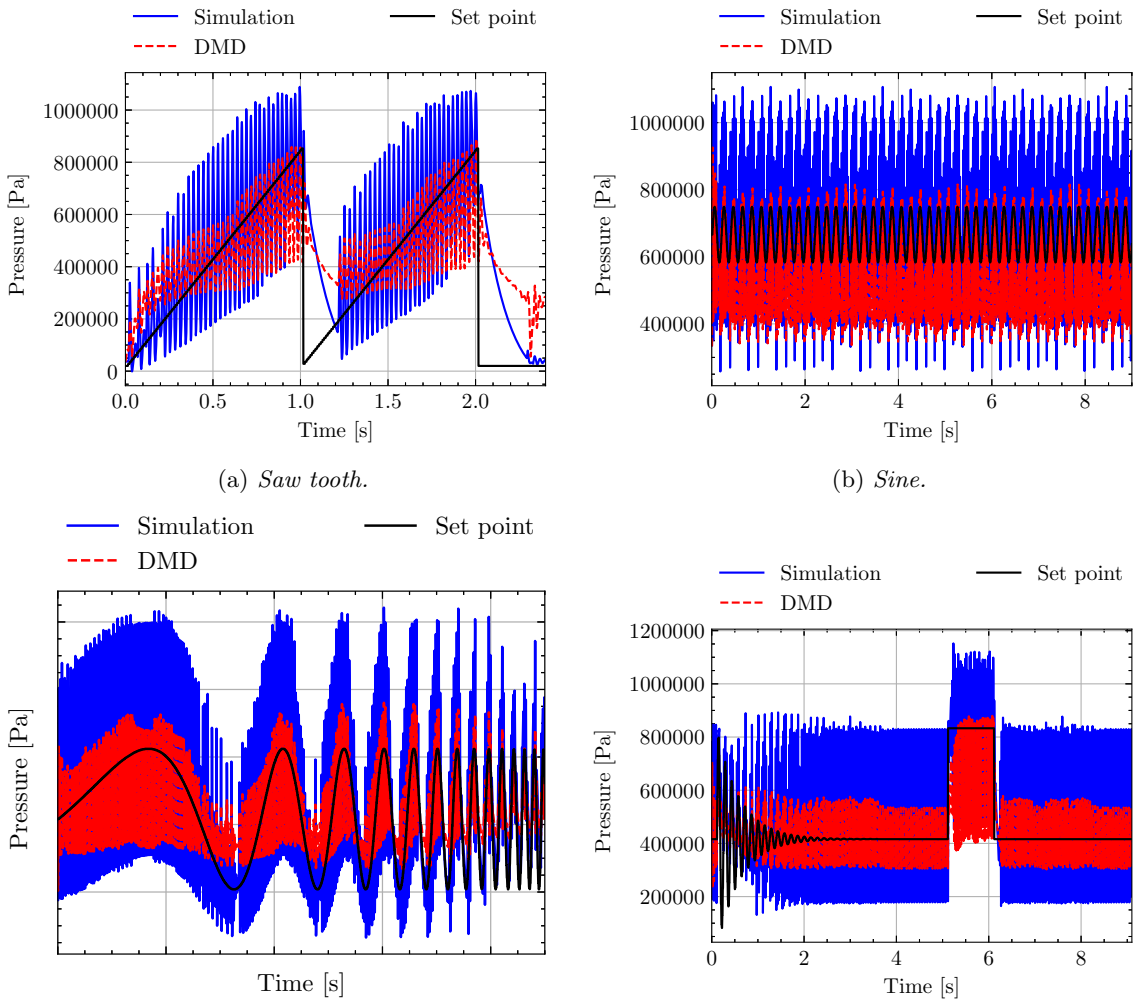


Figure 8.5: *Visualization of the growth/decay of the $n_r = 50$ modes at their respective frequency. Note that because the eigenvalues come in pairs that are complex conjugated, only $n_r/2 = 25$ data points are shown in this figure.*

8.1.3 Comparison with High-fidelity Simulations

In this section we compare the performance of the truncated DMD method described in Alg. 2 to a high-fidelity simulation. We first present the unfiltered solutions and then the filtered ones for the validation and test set. We measure the errors in absolute and relative terms as defined in Eq. 8.0.1-8.0.2. In addition, a Bode diagram of the filtered chirp solutions is presented.

Using the result from the previous section, which indicates that $n_r = 50$ is an appropriate truncation, we perform a DMD on the training data and then use it to compute the solution on all data sets. A comparison between the DMD solution and the simulated one on the validation and test set is shown in Fig. 8.6 with associated errors in Tab. 8.1. The comparison for the training data is presented in Appendix A. We first remind the reader that the blue and black solid lines, which display the simulated and the set point pressure respectively, are identical to the ones presented in Ch. 7. The red dashed lines represent the DMD solution and it is trained to follow the blue simulated solution. From this figure we make the following observations. Firstly, the DMD solution manages to capture the overall behaviour of the pressure signal well and it deviates less from the set point pressure than the simulated one. Even though it manages to reconstruct many of the frequencies of the signal, it can not fully capture the corresponding amplitudes. This behaviour is clearly seen in the solution of the chirp signal in Fig. 8.6c. Lastly, as seen in Fig. 8.6b and 8.6d, the DMD solution is also almost consistently offset compared to the simulated one.



(a) *Saw tooth.*

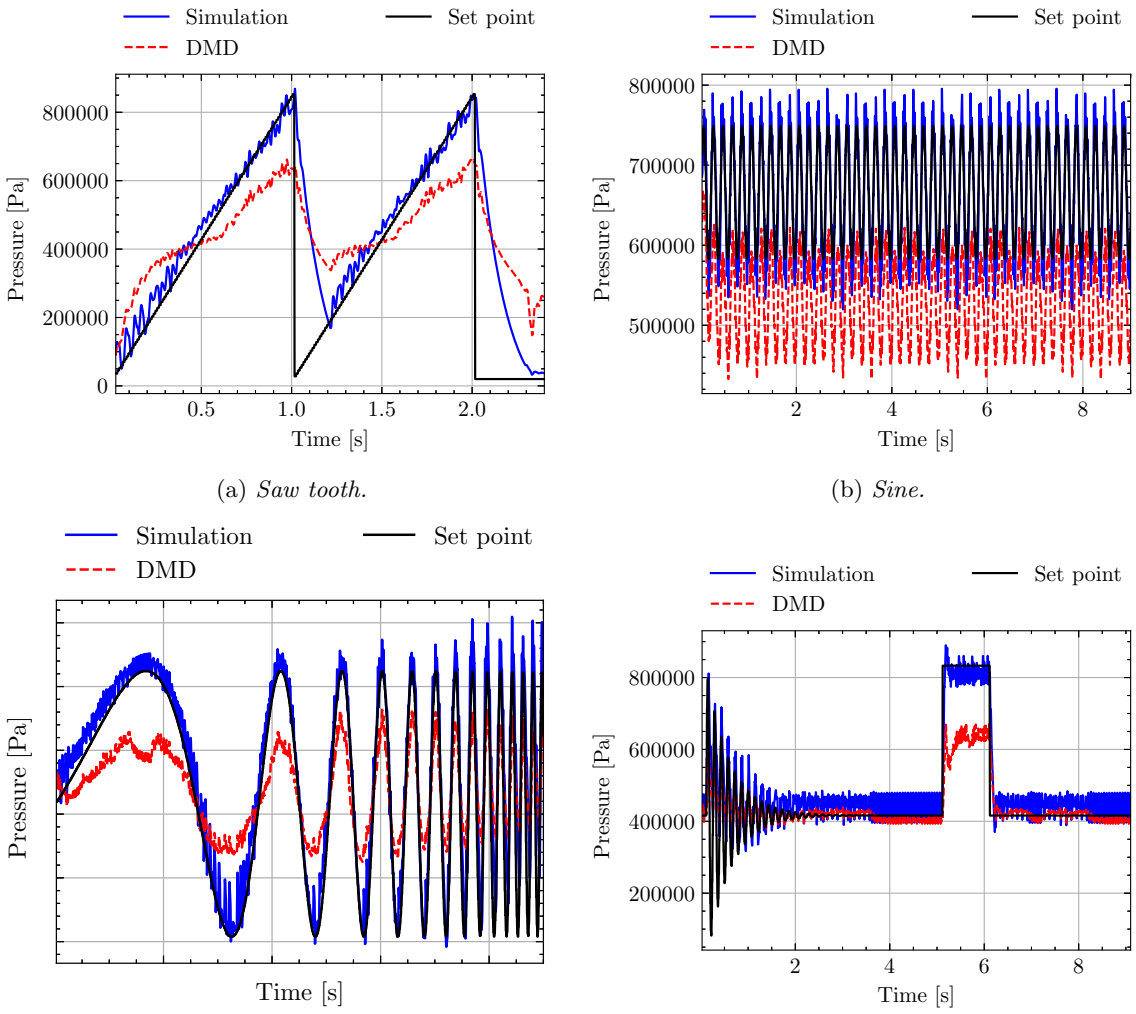
(b) *Sine.*

(c) *Chirp.*

(d) *Damped sine and pulse.*

Figure 8.6: *Comparison between unfiltered high-fidelity simulation and DMD predictions for (8.6a-8.6b) validation and (8.6c-8.6d) test data set using* $n_r = 50$. *See Tab. 8.1 for the numerical errors.*

We now apply a moving average filter to the results such that the dominant component at 37 Hz vanishes. The result from this filter is presented in Fig. 8.2b with corresponding errors in Tab. 8.1. Again, the figures of the solution to the training data set can be seen in Appendix A. In these post-processed solutions one can see that they deviate significantly less from the set point pressure than they did before. It is also evident that the simulated pressure resembles the set point pressure more than the pressure yielded by the DMD does. Furthermore, as expected, the absolute errors for the filtered DMD pressure are consistently lower than the unfiltered ones. One can also see that the relative error lies around 0.15 and 0.30 for the different data sets, except for the gentle ramp set where it is 1.00.



(a) *Saw tooth.*

(b) *Sine.*

(c) *Chirp.*

(d) *Damped sine and pulse.*

Figure 8.7: *Comparison between filtered high-fidelity simulation and DMD predictions for (8.7a-8.7b) validation and (8.7c-8.7d) test data set using* $n_r = 50$. *See Tab. 8.1 for the numerical errors.*

Table 8.1: *Absolute and relative errors between DMD solution and high-fidelity simulation for each data set.*

		Unfiltered		Filtered	
		ε_a [kPa]	ε_r	ε_a [kPa]	ε_r
Training	High pulse	7.63	0.365	6.75	0.320
	Low pulse	6.52	0.454	4.13	0.303
	Steep ramp	4.19	0.311	3.24	0.252
	Gentle ramp	4.85	0.939	4.57	1.00
	Tooth	3.92	0.356	2.68	0.257
	Trapezoid	3.37	0.297	2.67	0.244
	Sine	2.95	0.396	1.41	0.198
	Sum of sines	2.75	0.374	1.64	0.229
	Damped sine	3.50	0.384	1.32	0.153
Validation	Saw tooth	3.23	0.292	2.34	0.223
	Sine	1.83	0.250	1.46	0.207
Test	Chirp	1.49	0.268	0.918	0.177
	Damped sine and pulse	1.49	0.263	0.793	0.150

8.1.4 Bode Diagram

Another method for evaluation of the quality of the DMD predictions is the Bode diagram. A Bode diagram visualizes the frequency response of a system by considering how the magnitude and phase changes with increasing frequency [13]. This Bode plot was produced by finding the peaks of the chirped input and output pressure signal. The gain is then computed as the ratio of the peak-to-peak values of the actual and set point pressure and the phase as the argument of the gain. We used an existing script written by Haldex to perform these computations.

The Bode diagram of the simulated and predicted solution is shown in Fig. 8.8. As seen in Fig. 8.8a, the gain of the two solutions are significantly different but do experience a similar behaviour where it initially increases and then decreases. Looking at the predictions in Fig. 8.2b, this comes as no surprise. What is less obvious from the prediction figures are how well the phase of the DMD solution matches the simulated one, which is why a Bode diagram of the system was constructed. In Fig. 8.8b we can see that they are comparable with one another.

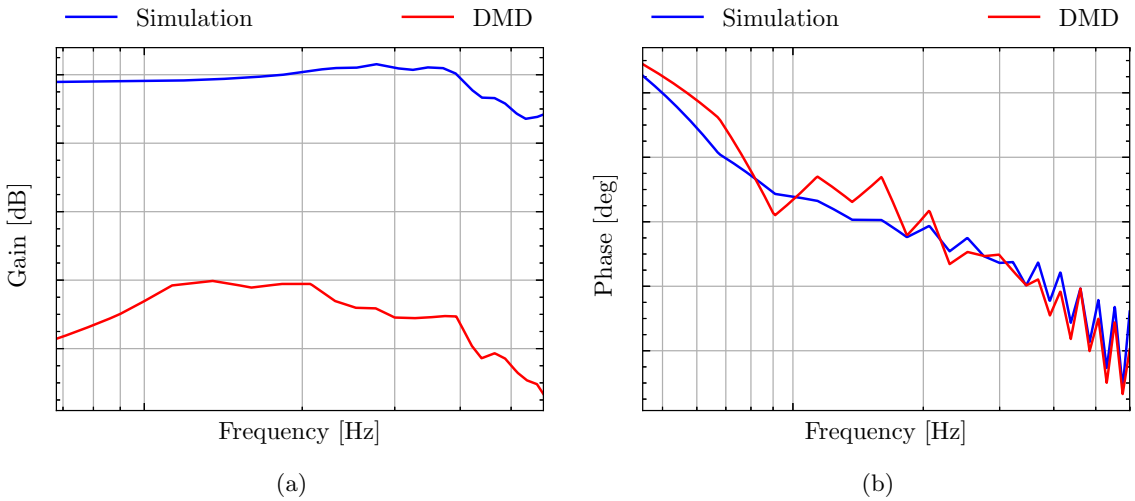


Figure 8.8: *Bode diagram of the simulated and the DMD solution of the FABV. Since the chirp signal was used to generate these figures, the scale on the axes has been removed. Note that the frequency axes are in logarithmic scale.*

8.1.5 Time Complexity

There are mainly two things that are of interest regarding the time complexity of predicting the system with the DMD method. One, how the prediction time scales with n_r. This result is presented in Fig. 8.9. Apart from the second sample point the line follows a linear trend with a slope of one.

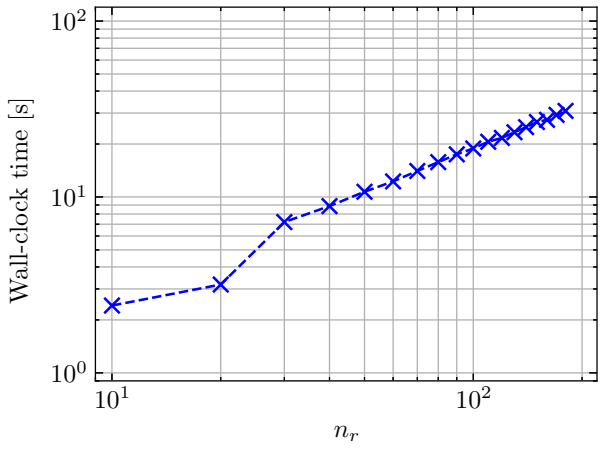


Figure 8.9: *Prediction time for the DMD method for different values of n_r.*

Two, how the prediction time of DMD compares to the simulation time of the FABV FMU. The closest comparison possible was to time predictions made with the DMD method using Python and NumPy and compare with the simulation time of the FMU using PyFMI and Dymola using a relative tolerance of 10^{-1}. This tolerance was set because it was the highest one possible in order for PyFMI and Dymola to compute a solution. PyFMI presents the wall-clock time spent in each FMU that was part of the co-simulation. The presented simulation time is therefore the simulation time of the FABV

FMU. In addition, the simulation time from Dymola is also presented as a comparison. Similar to PyFMI, Dymola presents the simulation time of each individual FMU. In Tab 8.2 the prediction time and the simulation times are presented. First and foremost, the prediction time of the DMD method is approximately 75% lower than both simulation times. Secondly, the simulation time with Dymola and PyFMI are comparable, which is expected.

Table 8.2: *Prediction time of DMD compared to the simulation time of the FABV FMU in both Dymola and PyFMI.*

	DMD	PyFMI	Dymola
Time [s]	10.6	40.7	39.1

8.1.6 Final Remarks

In this section we describe one additional test, namely when training the DMD on filtered data.

Note that when training and testing the DMD on filtered data we do not need to perform any post-processing of the result. We filter the data using the previously described shifted moving average filter as well as with a centred version of it. Precisely as before we found that $n_r = 50$ was suitable, which gave the relative errors on the test set presented in Tab. 8.3. Note that the errors when using unfiltered data are the same as presented in Tab. 8.1. The errors are well comparable with each other. Furthermore, the lowest errors were achieved using unfiltered data.

Table 8.3: *Absolute and relative errors on the test set when using the DMD method with filtered and unfiltered data. The data was filtered using a shifted and a centred moving average filter. The results when using unfiltered data have been filtered with a shifted moving average filter before the errors were computed, i.e. exactly as in Tab. 8.1.*

	Chirp		Damped sine and pulse	
	ε_a [kPa]	ε_r	ε_a [kPa]	ε_r
Shifted	9.29	0.180	8.00	0.151
Centred	9.54	0.184	8.24	0.156
Unfiltered	9.18	0.177	7.93	0.150

8.2 Koopman Method

In this section we present the predictions made with the Koopman method and the neural networks used for the vector-valued observable functions. The hyperparameters of the neural networks will also be presented together with the final model. Adam was used as the optimization algorithm, which is described in further detail in Sect 6.2.2. The exponential decay rates for the momentum estimates, ρ_1 and ρ_2, were left at their default values 0.9 and 0.999 respectively. The constant used for numerical stability δ was also left at its default value 10^{-7}.

For training, the Koopman modes were computed individually for each data set in the training set. This means that for each epoch, Koopman modes were calculated eight times. The data was normalized to zero mean and unit variance for each state. This normalization was done based on the training data, but applied to all data sets. A dual socket computer with two Intel Xeon Gold 6154 processors and 384 GB RAM was used for training.

For confidentiality reasons the chirp data set was left out from the test set for this section. In addition to that, the gentle ramp data, see Fig. 7.2d, was removed from the training set. This was done due to it causing the eigendecomposition, which is required to identify the Koopman modes, to fail.

8.2.1 Hyperparameter Tuning

Before we present the considered hyperparameters, we would like to remind the reader about the network structure that was used, see Sect. 6.3.2. Simply put, there are three fully connected layers with bias and ReLU activation functions after the first and second layer. The input dimension, i.e. the number of states, is denoted n_x and the output dimension, i.e. the number of real-valued observable functions, is denoted n. The hidden dimensions are all the same and are denoted d_{hidden}. It is also worth to recap the Koopman loss, Eq. (6.3.6), and the Adam algorithm, Eq. (6.2.4), since there are hyperparameters connected to these equations as well. For the Koopman loss, the number of step taken for the prediction loss, m_{pred}, is considered a hyperparameter. Regarding Adam, the tunable parameters are the learning rate l and of course the number of iterations.

This leaves five hyperparameters left to tune, n, d_{hidden}, m_{pred}, l and the number of epochs. Similar to for instance Ex. 6.4.3 the neural networks were pre-trained with the reconstruction loss in Eq. 6.3.3, which gives another two hyperparameters, namely pre-training learning rate l_{pre} and the number of pre-training epochs. In total there were seven parameters to tune, which was done with a uniform grid search.

Both learning rates were tuned over a span from 10^{-9} to 10^{-2}. The number of pre-training epochs were tuned in the interval from 0-100 and the number of epochs from 5-20. Regarding the output dimension of the neural network n and the hidden dimension d_{hidden}, they were both tested in an interval from 50 to 500. We tested m_{pred} with values ranging from 50 to m.

Generally, the hyperparameters did not change the prediction capabilities heavily. For instance the value of m_{pred} made almost no difference. The biggest effect was increased RAM usage, which was caused by large values for n and d_{hidden}. The learning rate was as expected an important parameter to tune. However, most learning rates resulted in an almost constant loss. This also made it meaningless to train for much more than 10 epochs. Depending on the learning rate, the resulting predictions became unstable or stayed almost constant. It is also worth mentioning that some combination of parameters resulted in the eigendecomposition not being successful.

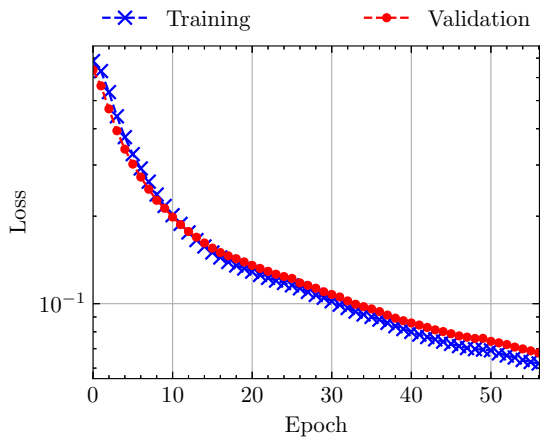
8.2.1.1 Final Model

The parameters for the final model were chosen based on the prediction error on the validation set. This resulted in the parameters found in Tab. 8.4.

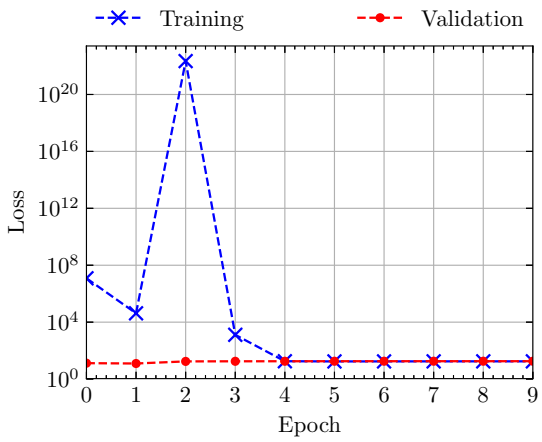
Table 8.4: *Hyperparameters used for the final neural networks.*

Hyperparameter	n	d_{hidden}	m_{pred}	l	l_{pre}	epochs	pre-epochs
Value	100	150	m	$3 \cdot 10^{-5}$	$4 \cdot 10^{-4}$	10	57

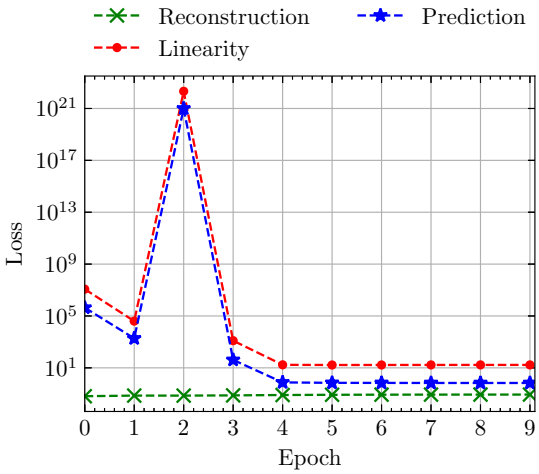
Fig. 8.10 shows how the training and pre-training losses varies with the number of epochs. It took approximately three hours to train the final model on the aforementioned hardware.



(a) *Pre-training loss.*



(b) *Total Koopman loss.*

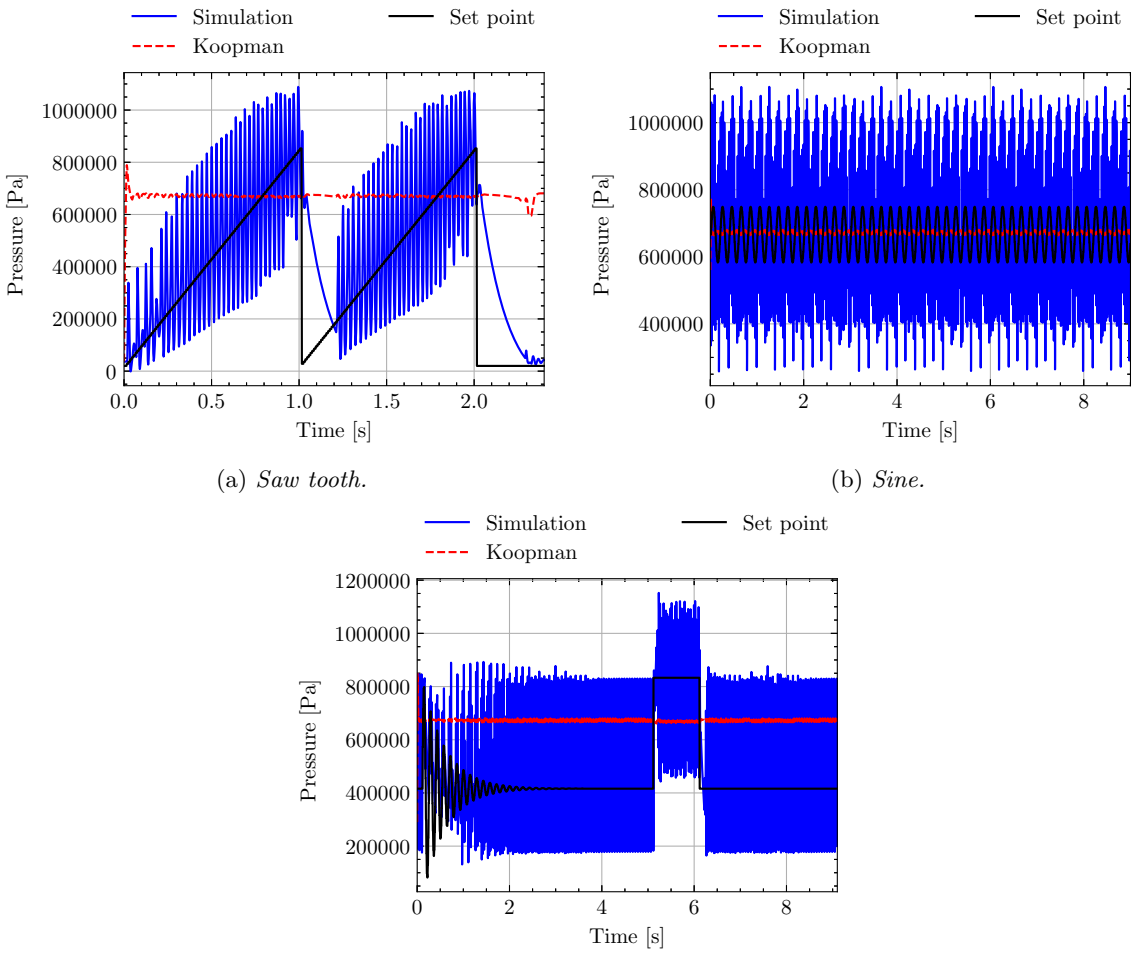


(c) *Individual training losses.*

Figure 8.10: *Loss from the training of the final model.*

8.2.2 Comparison with High-fidelity Simulations

In this section the result from the validation and test sets are presented. The predictions generated by the Koopman method together with the simulated solutions are shown in Fig. 8.11 with corresponding errors in Tab. 8.5. The result from the training set is presented in Appendix A. Once again the reader is reminded that the blue and black solid lines represent the simulated and the set point pressures, respectively. The red dashed line represents the predictions from the Koopman method.



(a) *Saw tooth.*

(b) *Sine.*

(c) *Damped sine and pulse.*

Figure 8.11: *Comparison between predictions made with the Koopman method and high-fidelity simulations. Fig. 8.11a-8.11b are from the validation set and Fig. 8.11c is from the test set. See Tab. 8.5 for the corresponding numerical errors.*

The Koopman method was unable to produce meaningful predictions. Regardless of input, the predicted pressure stays almost constant at around 670 kPa. The DMD method resulted in significantly more accurate results, which is made clear from the prediction errors in Tab. 8.5.

Table 8.5: *Absolute and relative errors between Koopman solution and high-fidelity simulations for each data set.*

		ε_a [kPa]	ε_r
Training	High pulse	12.5	0.597
	Low pulse	14.2	0.988
	Steep ramp	10.6	0.790
	Tooth	8.46	0.767
	Trapezoid	6.32	0.557
	Sine	2.54	0.342
	Sum of sines	2.27	0.308
	Damped sine	5.02	0.551
Validation	Saw tooth	7.44	0.673
	Sine	2.17	0.296
Test	Damped sine and pulse	3.05	0.539

8.2.3 Temporal Koopman Modes

Unlike the temporal DMD modes, the temporal Koopman modes are not directly related to the states. Instead they describe the dynamics of the observables rather than the states. This means that the observables evolution in time can be described in terms of the frequencies and exponential growth or decay of the Koopman modes. As a reminder, the observables are the output of the neural network, which implies that each observable is a function of all states. A plot of the temporal modes can be found in Fig. 8.12. The result is different from the temporal DMD modes, which is expected. Generally, most modes correspond to a frequency below 100 Hz. It is also worth pointing out that one eigenvalue has an absolute value greater than one, and is therefore unstable. However, none of the figures in Fig. 8.11 indicate any type of unstable behaviour.

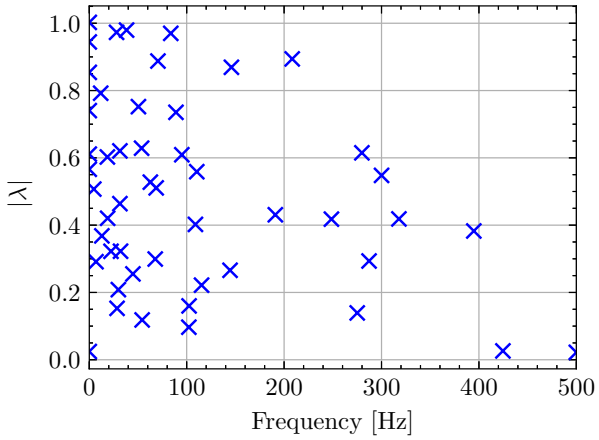


Figure 8.12: *Visualization of the growth/decay of the $n = 100$ modes at their respective frequency. Note that because the eigenvalues come in complex conjugated pairs, only $n/2 = 50$ data points are shown in this figure.*

8.2.4 Final Remarks

Even though the Koopman method did not produce usable predictions, it is still of interest to investigate how time consuming predictions made with the Koopman method are for our final model. The predictions were made using NumPy and TensorFlow. The prediction time is presented in Tab. 8.6 together with the result from Sect. 8.1.5 to remind the reader about the previous results. The prediction time of the Koopman method is significantly shorter than the ones of PyFMI and Dymola, but longer compared to the DMD method.

Table 8.6: *Prediction time of the Koopman method compared to DMD and the simulation time of the FABV FMU in both Dymola and PyFMI.*

	Koopman	DMD	PyFMI	Dymola
Time [s]	20.1	10.6	40.7	39.1

Chapter 9

Discussion

We now present a few discussion topics for each method as well as potential improvements and continuations.

9.1 Dynamic Mode Decomposition Method

We have shown that DMD manages to extract dynamical features of a system by decomposing data into low-dimensional spatial-temporal modes. The relevance of these system characteristics have also been validated by demonstrating how they can be used to evolve a system in time. In the case of a linear system this reconstruction is exact, whereas it for a non-linear system can be interpreted as a linear approximation. In other words, the DMD provides a computationally efficient linear model for how the most relevant dynamical components of a system evolve in time. This linear model can thus be seen as a reduced order model of a system.

Furthermore, we have demonstrated the capabilities of utilizing DMD in reduced order modelling by applying it to a highly non-linear system, namely the FABV. As seen in Fig. 8.7, the DMD method performs well with a relative error on the test set of about 0.16 after filtering. This amount of precision is considered to be satisfactory given that the method is about four times faster than simulating the high-fidelity model using sophisticated solvers. Worth mentioning is that the DMD method manages to capture the frequency of the solution well, as e.g. depicted by the Bode diagram in Fig. 8.8. The phase delays of the DMD and simulated solutions, when using the chirp signal as input, closely resemble each other. The same can not be said about the gain. The DMD method consistently struggles with predicting an accurate amplitude of the solution. It may however be possible to apply an amplifier to get more precise amplitudes.

If the characteristics of a given system do not vary in time, the DMD modes and eigenvalues only have to be computed once. The modes can then be stored and loaded into memory when needed. Once the modes are loaded and an initial value of the states are given, the modes can efficiently be used to evolve the system in time. Tab. 8.2 showed that this method could reduce the computing time of the FABV by approximatively 75%, when compared to simulations done using sophisticated solvers. The DMD method managed this whilst having a relative error of about 0.16 on the test set. This comparison is not completely fair since the relative tolerance of the solvers were set to 10^{-1}. This means that the solutions yielded by PyFMI and Dymola are more accurate than the DMD one. However, it was not possible to simulate the FABV with a relative tolerance larger than 10^{-1}. The DMD method thus has

the advantage of being able to compute a less accurate, but still meaningful, solution than currently possible with traditional simulation tools. Worth noting is that if the system characteristics were to significantly change in time, the DMD method would probably still be useful if it was given data from the different stages of the system. However, the number of modes needed to be able to capture this continuous change of dynamics would also increase significantly. To this end, Lusch *et al.* (2018) [20] proposed a framework in which a neural network was introduced that parameterized the continuous eigenvalue spectrum. This allowed for a more compact and efficient implementation.

Using truncated DMD, one can reduce the number of modes from n_x to only include the n_r most relevant ones. This can be especially useful when having large-scale recordings with $n_x >> m-1$, which often is the case of flow fields. In the case of the FABV, we were able to reduce the rank of X significantly without losing a notable amount of information, see Fig. 8.1. In theory, the reconstruction error should decrease with increasing number of modes. However, as seen in Fig. 8.2, this behaviour was not observed as the error plateaued around $n_r = 50$ modes. In conclusion, using truncated DMD we were able to get a reduced order model of the FABV using only a fraction of DMD modes compared to the number of states it has. It is worth noting that during the prediction phase, the time complexity of the method is $\mathcal{O}(n_r)$. This means that there exists a trade-off between desired accuracy and time consumption.

Next, we give a brief discussing regarding time delay embeddings. As seen in Fig. 4.8, by using time delay embeddings the DMD method was able to improve its performance significantly. It even managed to produce a prediction that was very well in line with a high-fidelity simulation of a highly non-linear system. This can be explained by the fact that the number of modes used, which is restricted to number of rows in X, were not enough to capture the non-linearities. Appending q time-shifted data points to the snapshots increases the number of rows in X from n_x to qn_x. The maximum possible number of DMD modes are thus increased by utilizing time delay embeddings. With $n_x = 4$ and using $q = 900$ the method was able to represent the non-linear dynamics using $qn_x = 900 \cdot 4 = 3600$ modes that could evolve the system linearly in time, thus producing an accurate solution. The reason why this technique was not considered for the FABV was due to the requirement of more than one initial value in order for the method to be able to start evolving the system. This would also make the reduced model incompatible if it were to be exported into an FMU and used in a co-simulation.

Worth noting is that when evolving the states of the system in time one could choose to not include all extracted spatial-temporal modes. One could e.g. sort the modes by their magnitude and only include a few of the highest ones, or e.g. only include modes with frequencies inside a certain frequency range. The effect of this is however not investigated further as it is considered to be outside the span of this thesis. We instead rely on the truncated DMD to provide the n_r most relevant modes and then simply use all of them when computing the solution of the system.

9.2 Koopman Method

Regarding the Koopman method, we have successfully applied it to smaller non-linear systems. In these cases we were able to learn suitable vector-valued observable functions resulting in the Koopman method predicting the states more accurately than the DMD method. For the larger FABV model, we were not able to produce a usable reduced order model. The predictions simply lacked accuracy and relevance. The produced model was however faster than a high-fidelity simulation.

To continue we would like to add a few comments on the hyperparameter tuning and the received losses. After that we will discuss some potential reasons behind why the training of the models were so time consuming as well ass some potential solutions. Potential improvements to the Koopman method will also be discussed.

One thing that heavily influenced how thoroughly the hyperparameters were tuned was the time it took to train the models. Just as a reminder, pre-training for 57 epochs and training with the Koopman loss for another 10 epochs took almost three hours. To be more specific, pre-training was fast and took roughly 15 min, but training with the Koopman loss was slow and took most of the remaining time. This of course impacted the tuning of the combination of learning rate and number of epochs. Another consequence of the slow training was that the individual scaling factors for the Koopman loss were simply set to one. There are likely some improvements to be found from tuning them individually judging by Fig. 8.10c, where especially the reconstruction loss remains constant. Batch size is a common hyperparameter that we also did not have time to tune, which potentially could have changed the result.

There are a few potential problems causing the slow training. To begin with, memory usage. During training of our final model we saw upwards of 70 GB of RAM usage. For models that contained larger weight matrices, i.e. where n and d_{hidden} were bigger, the RAM usage exceeded 300 GB. Since memory bandwidth is limited and memory allocation generally can be considered slow, training such large models were very time consuming. In an attempt to reduce the memory usage, all the data was stored with single-precision instead of double-precision. This did indeed reduce memory usage and speed up training, but the lost numerical precision led to unsuccessful eigendecompositions instead. One potential remedy that was not tested is to exclude some of the states. At least 800 states are namely constant. By excluding 800 states from the state vector $\mathbf{x}$, the number of weights in the input layer of $\mathbf{g}$ and output layer of $\mathbf{h}$ would decrease with more that 30%. Since these layers contain almost 90% of the weights, this reduction would be significant. Furthermore, a network consisting of only fully connected layers quickly becomes large in the sense that it contains many parameters. This gives several high-dimensional gradients for the backwards pass. A dimensionality reduction of $\mathbf{x}$ would improve this situation as well, since fewer weights means lower dimensional gradients and fewer weights to update. Finally, we were not able to use GPU accelerated training. This limitation comes from TensorFlow, since there is no support for handling complex numbers on the GPU in the backwards pass.

In theory the Koopman method can be more capable than the DMD method, given wisely chosen observable functions. But, the predictions from the Koopman method of the FABV, see Fig. 8.11, did not come close to the accuracy of the DMD method. There is on the other hand room for improvements in a few different areas. Firstly, the data. This is most likely the biggest reason as to why the resulting predictions look like they do. In Fig. 7.2 all the data from the training set is presented. Remember that the gentle ramp was removed for the Koopman model. This leaves eight data sets, where two are pulses, three are ramps and three are sine-like signals. This may seem fairly balanced, but the key thing to note here is the length of each data set. Two of the sine-based inputs are nine seconds long. Together, those two contain almost two thirds of the total number of snapshots for the training set. The same thing holds for the validation set, where again two thirds of the total number of snapshots come from sine-like signals. Therefore, an easy way to provide a low loss is to find the mean value of the sine-based data sets, which it seems like is exactly what has happened. This then gives a low loss on both the training set and validation set as well as low prediction error on the validation set.

Secondly, the metric used for selecting hyperparameter values. As stated before, hyperparameters were chosen based on prediction error for the pressure on the validation set. Even though the pressure is what we are interested in, the other states still matter, since they are used for predictions of the pressure. Thirdly, the use of complex numbers. To clarify, TensorFlow does not support complex number as weights in a neural network. When a complex-valued vector is passed through a real-valued neural network, the complex part of the input is completely ignored. The problem is that the Koopman modes can be complex-valued. Therefore, potential information is lost when transforming observables back to the states. Finally, network architecture. In this thesis the network architecture was limited to a basic encoder and decoder with a few fully connected layers. There are many other network

architectures that probably could make sense for the problem at hand, for instance recurrent neural networks (RNNs). As stated in the previous section, Lusch *et al.* [20] propose a third neural network used to update the Koopman modes in time. Another option would be to replace the Koopman operator entirely by a neural network, similar to what was presented by Otto and Rowley in [21]. In addition to that, there are of course other types of layers that could be used instead of just fully connected layers.

With all this said, there are two reasons that point towards that the Koopman method could produce more accurate and useful predictions. Firstly, the predictions presented in Ex. 6.4.3. The Koopman method manages to produce predictions with a significantly lower prediction error than the DMD method. Secondly, there are plenty of areas that were not completely covered in this thesis. With more time for tuning and more data that also is better balanced, the result could potentially be improved significantly. Since the prediction time was lower than the high-fidelity simulations, we consider it relevant to explore these areas more.

9.3 Future Work

An interesting follow-up to this thesis would be to export our model from the DMD method to an FMU. This would allow for a more accurate comparison between the DMD method and the simulation tools in regards to time complexity. It would also make it possible to test the model in its intended use case, i.e. to be used in a FMI compatible simulation environment. On the topic of DMD, there are still other types of DMD algorithms that could be explored. Not only that, constrained optimization, such as ridge regression or LASSO, could also be of interest. Then there is the data. Even though an investigation of the impact of the data probably would be targeted at the Koopman method, it would most likely affect the DMD result as well. As mentioned previously, we believe that this truly could make or break the neural network-based observable functions. To be more specific in terms of more data, the different data sets should probably be of different amplitude, but crucially, the overall balance has to be considered. When it comes to network architecture the options are endless. There are some specific ideas that others have used successfully, such as RNNs and neural network representations of the Koopman operator. Investigating these options would be a suitable continuation of this theses.

Chapter 10

Conclusion

In this thesis, we have investigated two potential frameworks for model order reduction, which are based on modal decomposition of data, namely using dynamic mode decomposition (DMD) and Koopman spectral analysis. We have validated the technique for extracting dominant dynamical characteristics from data for reduced order modelling by applying the methods to both linear and non-linear systems. When applied to a model of a large highly non-linear system, our results show that the DMD method outperforms the Koopman method on unseen data, both in accuracy and in time consumption. More specifically, the DMD solution managed to reconstruct both the low and high frequencies of the system well, but lacked precision in terms of amplitude and offset. This resulted in a relative error of 0.16 and a computational time four times shorter than that of a sophisticated solver with a relative tolerance of 0.1. Moreover, we were unable to get a meaningful solution to the system using the Koopman method. The main reason behind this presumably derives from using non-representative data when training the neural networks. Worth mentioning is however that our Koopman implementation was successfully applied to a smaller non-linear system and was notably faster than simulating the system using the aforementioned solver.

In conclusion, a reduced order model of a large highly non-linear system with meaningful fidelity was produced using the DMD method. Additionally, this reduced model can be simulated significantly faster than the original high-fidelity model. The DMD method therefore shows great promise in being used in a reduced order modelling framework. Lastly, we believe that proposed improvements to the Koopman method would result in it being able to obtain a meaningful representation of the system.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Christian Andersson, Claus Führer, and Johan Åkesson. "Assimulo: A unified framework for ODE solvers". In: *Mathematics and Computers in Simulation* 116 (2015), pp. 26 –43. ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2015.04.007.

[3] Christian Andersson, Johan Åkesson, and Claus Führer. *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*. eng. Tech. rep. LUTFNA-5008-2016. Centre for Mathematical Sciences, Lund University, 2016. URL: https://lup.lub.lu.se/search/ws/files/7201641/pyfmi_tech.pdf.

[4] Modelica Association. *Functional Mock-up Interface for Model Exchange and Co-Simulation*. Oct. 2019. URL: https://github.com/modelica/fmi-standard/releases/download/v2.0.1/FMI-Specification-2.0.1.pdf.

[5] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.4*. Apr. 2017. URL: https://www.modelica.org/documents/ModelicaSpec34.pdf.

[6] George D. Birkhoff. "Proof of the Ergodic Theorem". In: *Proceedings of the National Academy of Sciences* 17.12 (1931), pp. 656–660. ISSN: 0027-8424. DOI: 10.1073/pnas.17.2.656.

[7] Torsten Blochwitz et al. "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models". eng. In: *Proceedings of the 9th International Modelica Conference*. The Modelica Association, 2012, pp. 173–184. ISBN: 978-91-7519-826-2. DOI: 10.3384/ecp12076173.

[8] Christoph Boeddeker et al. *On the Computation of Complex-valued Gradients with Application to Statistically Optimum Beamforming*. 2017. arXiv: 1701.00392 [cs.NA].

[9] Comsol. *Eigenfrequency Analysis*. 2018. URL: https://www.comsol.se/multiphysics/eigenfrequency-analysis (visited on 05/30/2020).

[10] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (July 2011), pp. 2121–2159.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016, pp. 163,223–230,286,288,301–302.

[12] Jonathan H. Tu et al. "On dynamic mode decomposition: Theory and applications". In: *Journal of Computational Dynamics* 1.2 (2014), 391–421. ISSN: 2158-2505. DOI: 10.3934/jcd.2014.1.391.

[13] Tore Hägglund. *Föreläsningsanteckningar Reglerteknik AK*. Institutionen för Reglerteknik Lunds Tekniska Högskola, 2015, pp. 29–30.

[14] Geoffrey Hinton and Tijmen Tieleman. *Lecture 6e - rmsprop: Divide the gradient by a running average of its recent magnitude*. 2012. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[15] Anders Holst and Victor Ufnarovski. *Matrix Theory*. Studentlitteratur AB, 2014, p. 384. ISBN: 978-91-44-10096-8.

[16] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251 –257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T.

[17] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359 –366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[18] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[19] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861 –867. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80131-5.

[20] Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. "Deep learning for universal linear embeddings of nonlinear dynamics". In: *Nature Communications* 9.1 (Nov. 2018). ISSN: 2041-1723. DOI: 10.1038/s41467-018-07210-0.

[21] Samuel E. Otto and Clarence W. Rowley. *Linearly-Recurrent Autoencoder Networks for Learning Dynamics*. 2017. arXiv: 1712.01378 [math.DS].

[22] K. B. Petersen and M. S. Pedersen. *The Matrix Cookbook*. Version 20121115. Nov. 2012. URL: http://www2.compute.dtu.dk/pubdb/edoc/imm3274.pdf.

[23] Joshua L. Proctor and Philip A. Eckhoff. "Discovering dynamic patterns from infectious disease data using dynamic mode decomposition." In: *International Health (1876-3413)* 7.2 (2015), p. 139. ISSN: 18763413.

[24] Clarence W. Rowley et al. "Spectral analysis of nonlinear flows". In: *Journal of Fluid Mechanics* 641 (2009), 115–127. DOI: 10.1017/S0022112009992059.

[25] Peter J. Schmid. "Dynamic mode decomposition of numerical and experimental data". In: *Journal of Fluid Mechanics* 656 (2010), 5–28. DOI: 10.1017/S0022112010001217.

[26] Naoya Takeishi, Yoshinobu Kawahara, and Takehisa Yairi. "Learning Koopman Invariant Subspaces for Dynamic Mode Decomposition". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 1130–1140. URL: http://papers.nips.cc/paper/6713-learning-koopman-invariant-subspaces-for-dynamic-mode-decomposition.pdf.

[27] TensorFlow. *tensorflow*. 2020. URL: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/linalg_grad.py\#L648 (visited on 04/19/2020).

[28] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*. SIAM, 1997, p. 35. ISBN: 9780898713619.

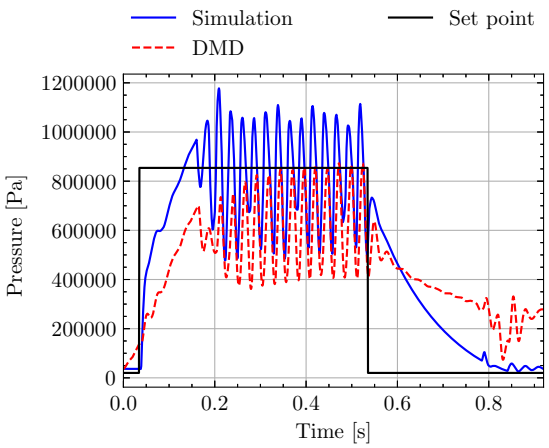
Appendix A

Result from Training Data Set

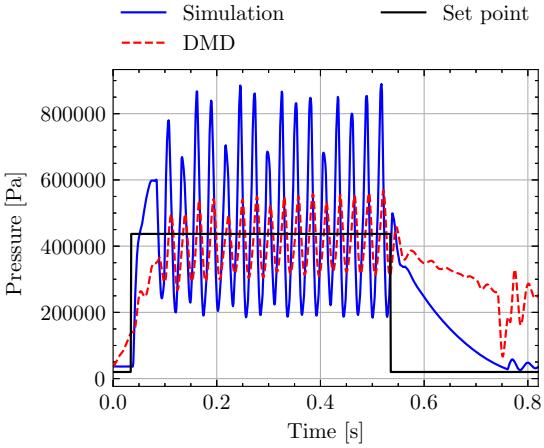
The predicted solutions for the training set are presented in this chapter, both from the DMD and the Koopman method.

A.1 Dynamic Mode Decomposition Method

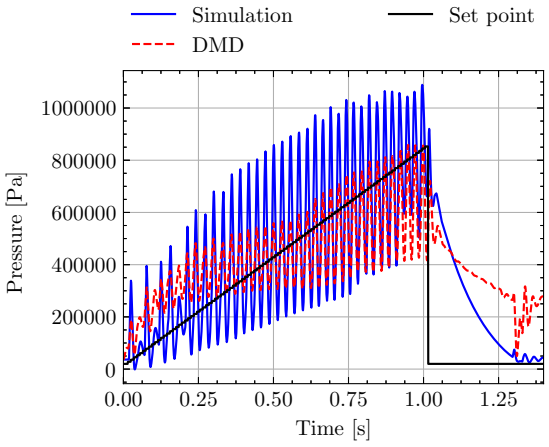
Fig. A.1 and Fig. A.1 show the unfiltered and filtered predictions made with the DMD method together with high-fidelity simulations respectively.



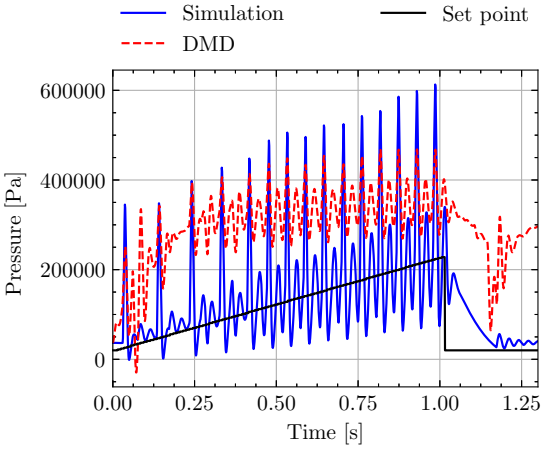
(a) *High pulse.*



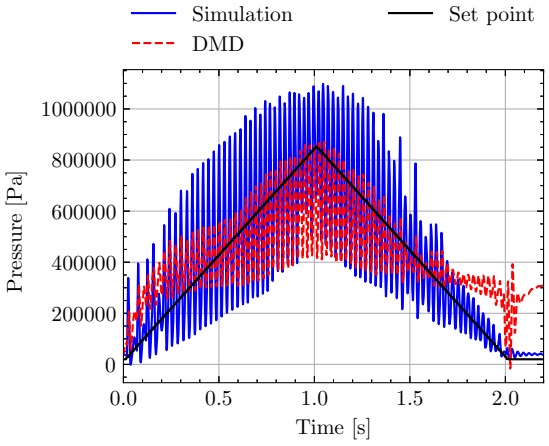
(b) *Low pulse.*



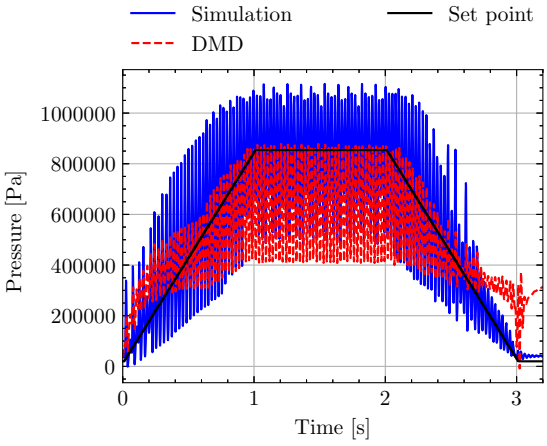
(c) *Steep ramp.*



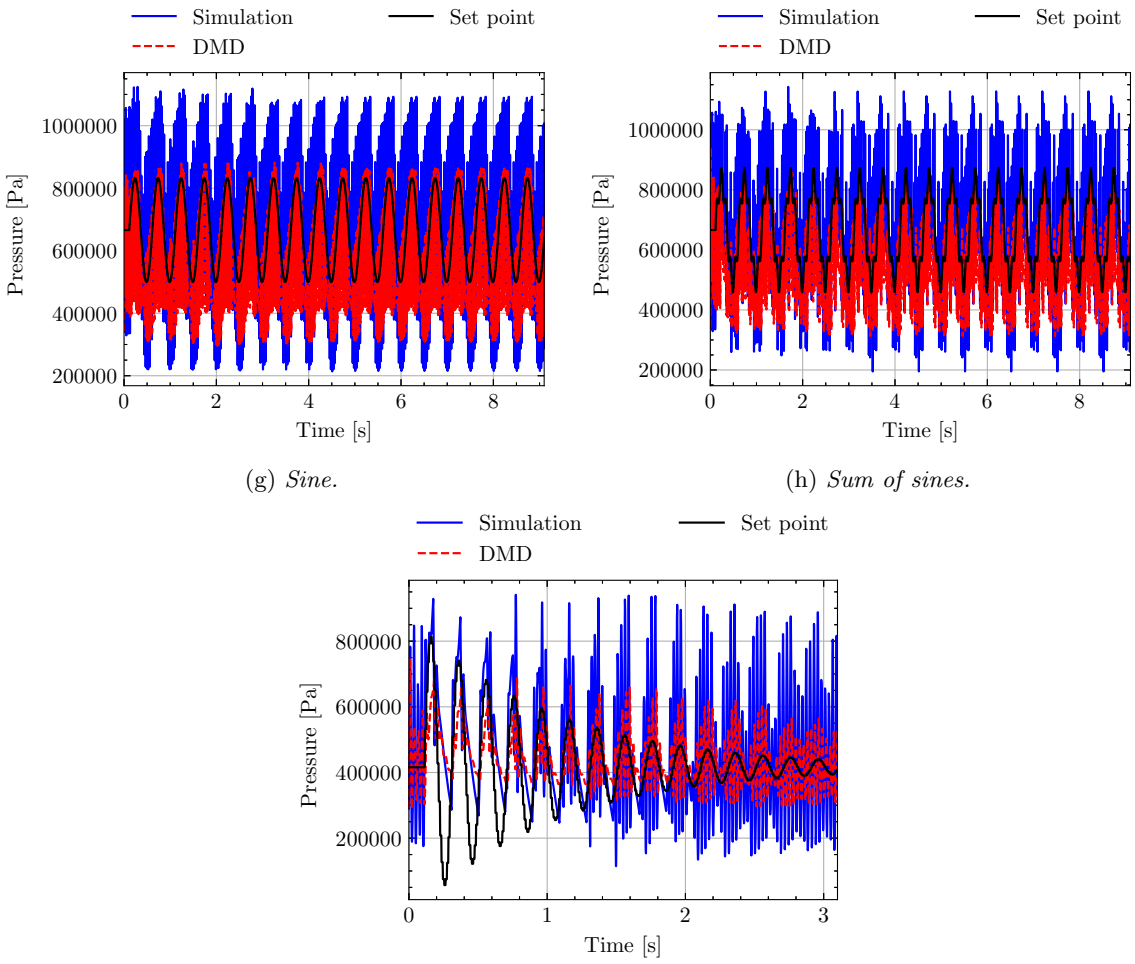
(d) *Gentle ramp.*



(e) *Tooth.*



(f) *Trapezoid.*

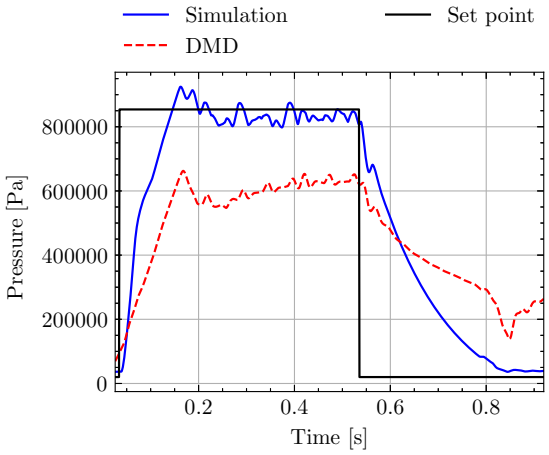


(g) *Sine.*

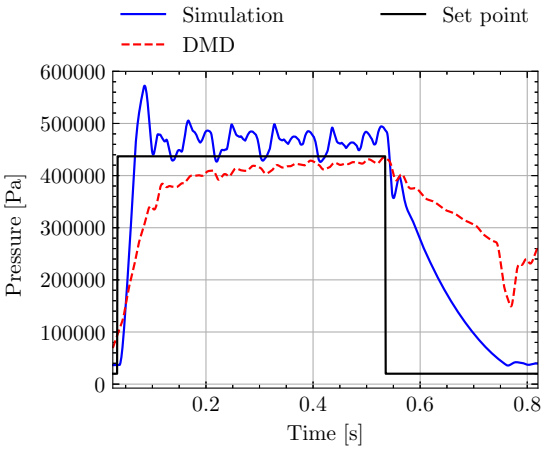
(h) *Sum of sines.*

(i) *Damped sine.*

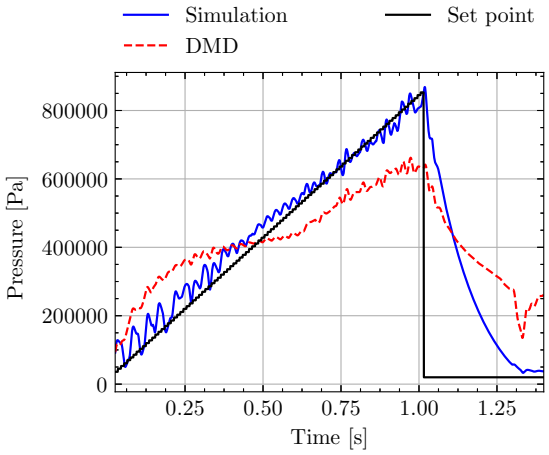
Figure A.1: *Comparison between unfiltered high-fidelity simulation and DMD predictions for the training data set. See Tab. 8.1 for the numerical errors.*



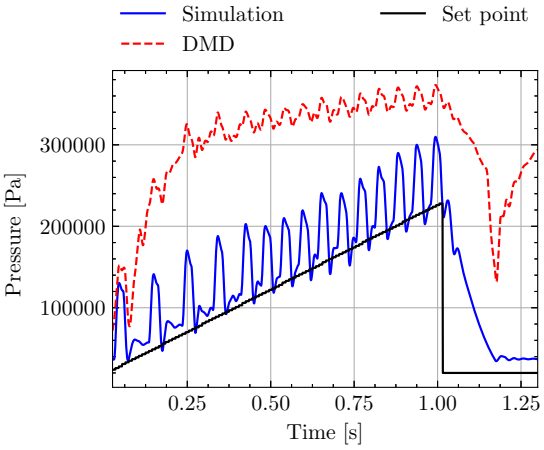
(a) *High pulse.*



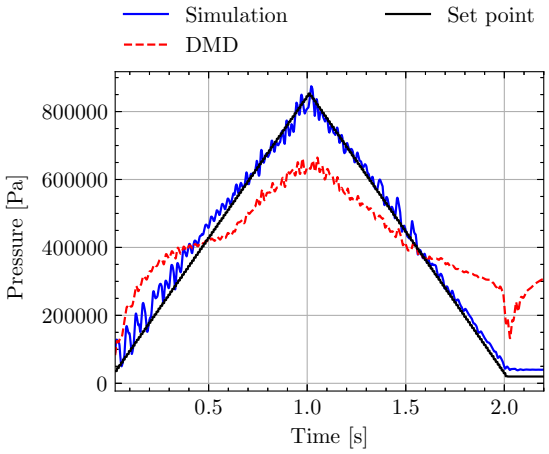
(b) *Low pulse.*



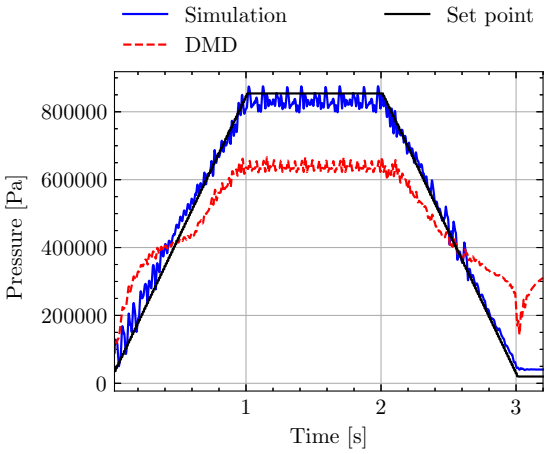
(c) *Steep ramp.*



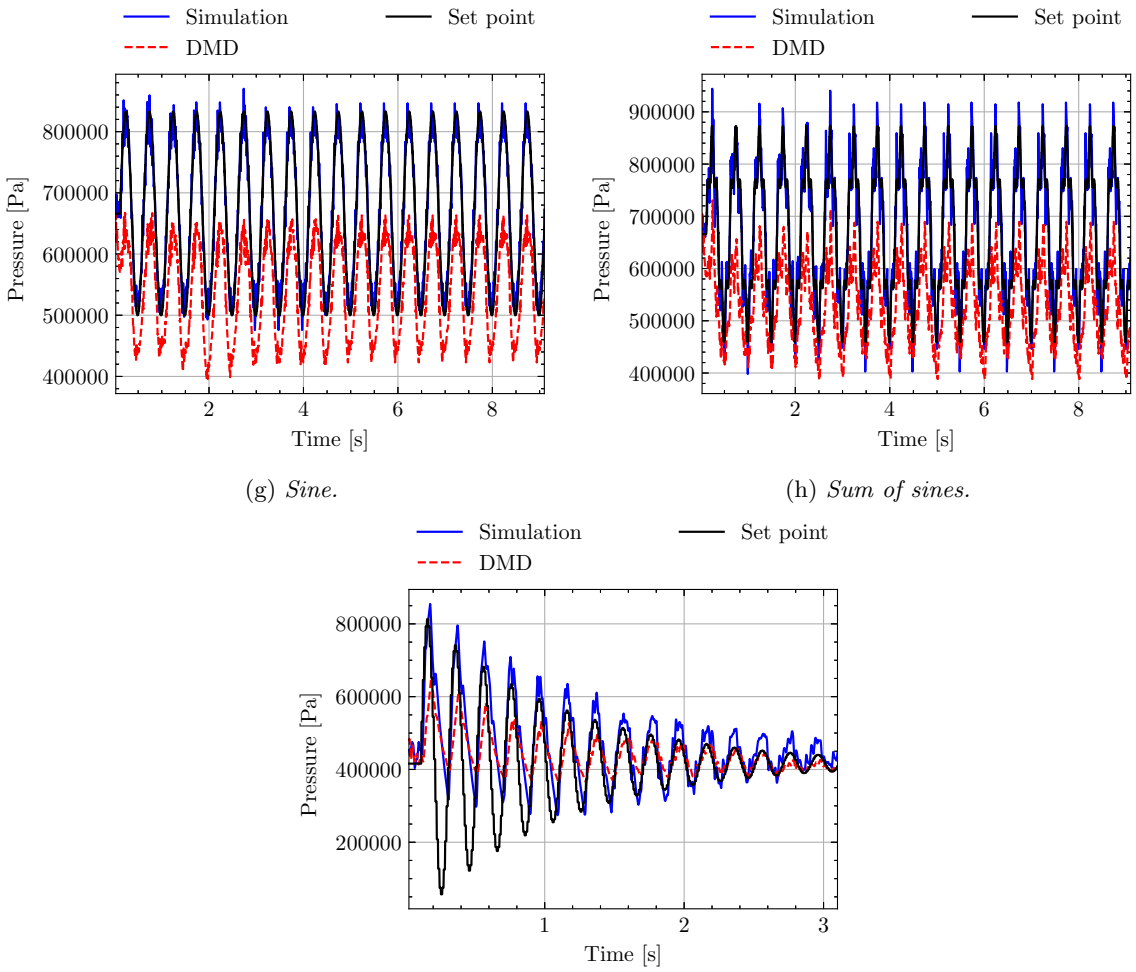
(d) *Gentle ramp.*



(e) *Tooth.*



(f) *Trapezoid.*



(g) *Sine.*

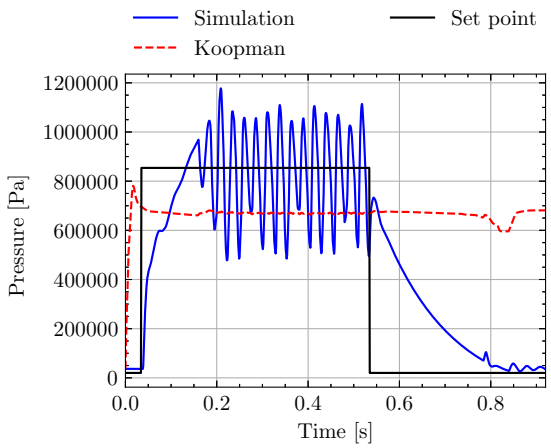
(h) *Sum of sines.*

(i) *Damped sine.*

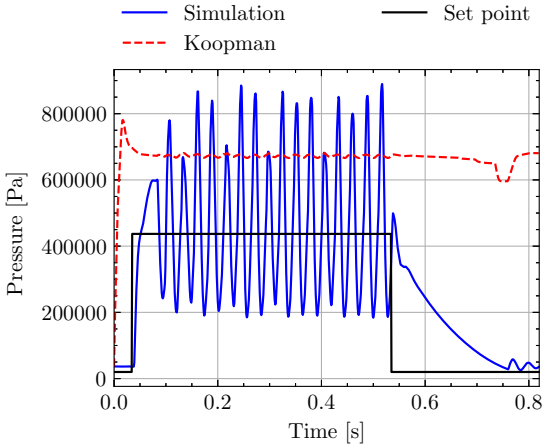
Figure A.2: *Comparison between filtered high-fidelity simulation and DMD predictions for the training data set. See Tab. 8.1 for the numerical errors.*

A.2 Koopman Method

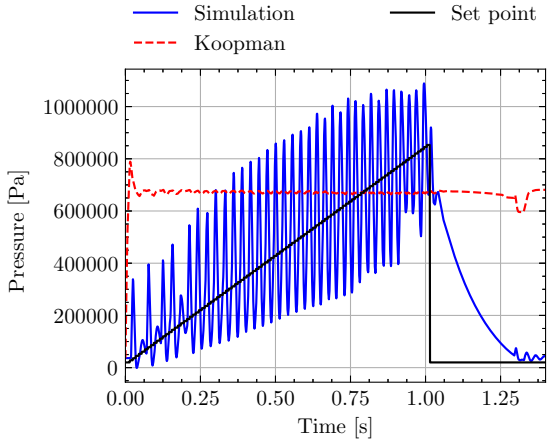
Fig. A.3 shows the predictions made with the Koopman method together with high-fidelity simulations.



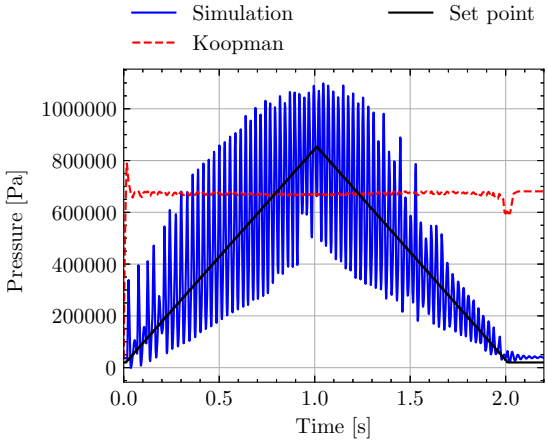
(a) *High pulse.*



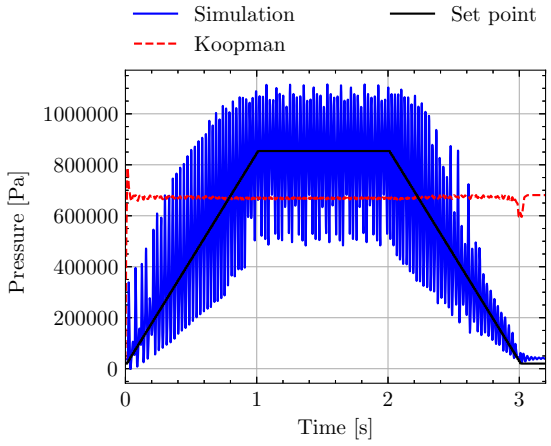
(b) *Low pulse.*



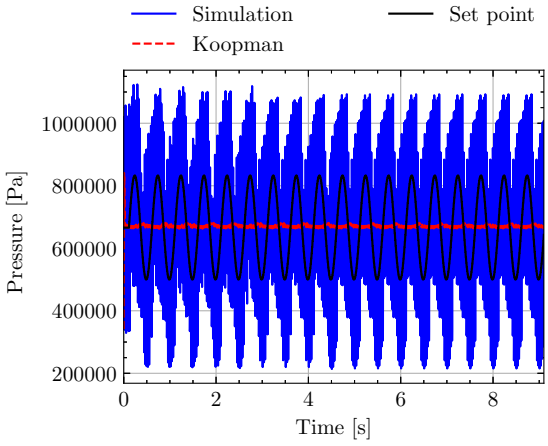
(c) *Steep ramp.*



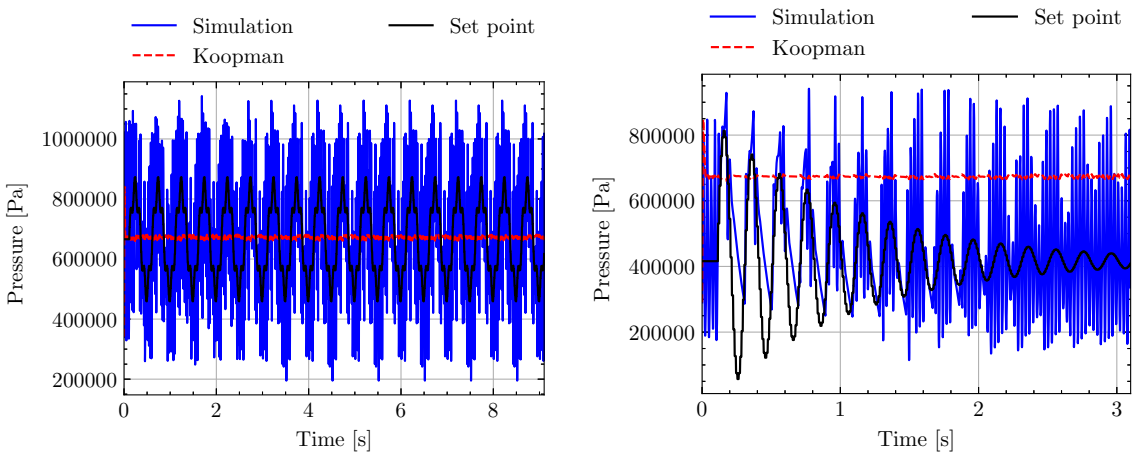
(d) *Tooth.*



(e) *Trapezoid.*



(f) *Sine.*



(g) *Sum of sines.* (h) *Damped sine.*

Figure A.3: *Comparison between predictions made with the Koopman method and high-fidelity simulations. See Tab. 8.5 for the corresponding numerical errors.*

Master's Theses in Mathematical Sciences 2020:E45
ISSN 1404-6342

LUTFNA-3048-2020

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/