Fakultät für Informatik
Technische Universität München

Operator-informed machine learning: Extracting geometry and dynamics from time series data

Daniel Lehmberg

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende:
Prof. Dr. Susanne Albers

Prüfende der Dissertation:
1. Prof. Dr. Hans-Joachim Bungartz
2. Prof. Dr. Gerta Köster,
   Hochschule München

Die Dissertation wurde am 21.12.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 04.03.2022 angenommen.

Acknowledgments

Writing my thesis was a challenging and rewarding journey. The struggles to learn peculiar mathematical concepts were followed by happy moments of understanding. I thank the people who accompanied and supported me throughout this journey. Firstly, I owe many thanks to my supervisors Gerta Köster and Hans-Joachim Bungartz for giving me the opportunity to undertake a PhD. I never would have thought that this would be a possibility for me. You were both always very supportive, giving me the freedom to explore new ideas, but also guiding me to make sure I did not get lost in them. I am also very grateful to Felix Dietrich, who had a big impact on my thesis. I enjoyed our time together in the US, discussing initial ideas of the thesis topic (and the many discussions since). My PhD experience was also much more enjoyable thanks to the companionship of the pedestrian research group at the Hochschule München. Thank you to Bene Zönnchen, Bene Kleinmeier, Marion Gödel, Christina Mayr, Stefan Schuhbäck, and Simon Rahn for the laughter and good times. I also thank Yannis Kevrekidis for hosting my research stay at Johns Hopkins University and for his ongoing interest in my work. I would like to express my deep gratitude to Lizzy Gosling for her endless patience, support and for proofreding my thesis. I am looking forward to exciting journeys to come. Finally, I thank my family for always supporting me.

The degree to which we connect to a community
is in proportion to our individuality.

— Rodney Mullen,
Godfather of street skateboarding

Abstract

Machine learning has become an established practice of scientific modeling, making use of ever-increasing data availability. In system identification a model extracts the dynamics of a system from time series data. For this task, however, many machine learning methods face serious limitations. Moreover, a complex model structure often obstructs further insight into the model's characteristics and the underlying system.

This thesis focuses on an operator-based approach to help overcome these limitations. Two linear operators are central to my thesis: the Laplace-Beltrami and Koopman operator. Both can extract, store, and describe essential information from time series data. The Laplace-Beltrami operator describes the geometry, whereas the Koopman operator captures the dynamics of the identified system. I also use time delay embedding to reconstruct partially observed system dynamics. This thesis aims to combine these three components into a single model and perform operator-informed system identification.

To explore the approach, I draw on three concrete data applications. In all three examples I show that the model accurately estimates the system and, moreover, that the approximated operators give valuable information to understand and interpret the model. First, I investigate a pendulum system to compare the model against known equations. The next two datasets stem from pedestrian dynamics and have no governing equations. For the second scenario, I build an efficient surrogate model to capture the macroscopic state dynamics that originate from a microscopic pedestrian simulation. In the third and most challenging scenario, I analyze data from pedestrian traffic in Melbourne, Australia. I demonstrate that the modeling approach captures the diverse traffic patterns and is robust to difficulties often faced in real-world settings.

Scientific software is essential for data-driven modeling. However, suitable software is lacking for my operator approach. Within my thesis I develop *datafold* as an innovative scientific software to unify the numerical frameworks (Diffusion Maps and Extended Dynamic Mode Decomposition) required for the operator approximation. *Datafold* covers the entire system identification workflow to promote flexible modeling and algorithmic experimentation.

By contributing scientific software and the analysis of concrete data scenarios, I enable new research possibilities to model dynamical systems from time series. I show how the approach advances a deeper understanding of systems, for which complex and inaccessible models are often adopted.

Keywords: system identification, data-driven modeling, scientific software, dynamic mode decomposition, diffusion maps, time delay embedding

Zusammenfassung

In der wissenschaftlichen Modellierung hat sich das maschinelle Lernen aufgrund stetig wachsender Datenverfügbarkeit etabliert. Bei der Systemidentifikation extrahiert ein Modell die Dynamik aus Zeitreihen. Dabei stoßen jedoch viele Methoden des maschinellen Lernens an Grenzen. Eine tiefere Einsicht in die Modelleigenschaften und das zugrunde liegende System werden oft durch eine komplexe Modellstruktur erschwert.

Diese Dissertation visiert einen operatorbasierten Ansatz an, um die Einschränkungen zu überwinden. Zwei lineare Operatoren stehen im Fokus meiner Arbeit: der Laplace-Beltrami- und der Koopman-Operator. Beide eignen sich, um wesentliche Informationen aus Zeitreihen zu extrahieren, zu speichern und zu beschreiben. Der Laplace-Beltrami Operator erfasst die Geometrie, während der Koopman Operator die Dynamik eines identifizierten Systems beschreibt. Zudem verwende ich Zeiteinbettungen, um eine unvollständige Systemdynamik in den Daten zu rekonstruieren. Das Ziel dieser Arbeit ist es, diese drei Komponenten in einem Modell zur Systemidentifikation zu kombinieren.

Um den Ansatz zu untersuchen, wähle ich drei konkrete Anwendungen. In allen Fällen zeige ich, dass das Modell die Systemdynamik genau beschreiben kann. Darüber hinaus liefern die approximierten Operatoren wertvolle Informationen, die sich zum Verständnis und zur Interpretation des Modells eignen. Im ersten Fall untersuche ich ein Pendelsystem, um das Modell anhand bekannter Gleichungen zu untersuchen. Die anderen beiden Datensätze stammen aus der Fußgängerdynamik und sind nicht mit Gleichungssystemen beschrieben. Im zweiten Fall entwickle ich ein effizientes Ersatzmodell, um die makroskopische Zustandsdynamik einer mikroskopischen Fußgängersimulation zu erfassen. Im dritten und anspruchsvollsten Fall analysiere ich Sensordaten zum Fußgängerverkehr in Melbourne, Australien. Der Modellierungsansatz kann die unterschiedlichen Verkehrsmuster identifizieren und ist gegenüber den Herausforderungen von Realdatenanalysen robust.

Für den von mir verfolgten operatorbasierten Ansatz mangelt es an wissenschaftlicher Software, welche für die datengetriebene Modellierung unverzichtbar ist. Im Rahmen meiner Dissertation entwickle ich deshalb die Software *datafold*. Dabei vereinige ich die numerischen Verfahren zur Approximation der Operatoren (Diffusion Maps und Extended Dynamic Mode Decomposition) in einem innovativen Framework. Die Software deckt den gesamten Ablauf der Systemidentifikation ab, so dass eine flexible Modellierung und algorithmische Experimente möglich werden.

Mein Beitrag zur Software und konkreter Analysen treibt die neuen Forschungsansätze zur datengetriebenen Modellierung von dynamischen Systemen voran. Ich zeige, dass der operatorbasierte Ansatz ein tieferes Systemverständnis eröffnet. Auch in Fällen, in denen oftmals komplexe und unzugängliche Modelle verwendet werden.

Contents

1 Introduction

The observation, experimentation, and analysis of systems that change over time are integral to many scientific fields. Seemingly unrelated subjects from diverse areas can be described in the generic framework of a *dynamical system*, comprising a state and its evolution over time. This includes well-known physical systems such as a pendulum or planetary motions, or systems with complex interactions such as Earth's climate [Froyland et al., 2021], epidemic outbreaks [Chowell, 2017], or crowd movements [Helbing et al., 2007]. Dynamical systems can also describe more abstract phenomena, such as the iterative learning progress of an artificial neural network [Dietrich et al., 2020]. A common goal in science is to develop a formal description of a system within a model:

> *By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work — that is correctly to describe phenomena from a reasonably wide area.*
>
> — John von Neumann, 1955

Within a traditional *equation-driven* workflow, scientists create models by experimenting within the system under well-controlled conditions, from which they can derive explicit equations. For example, Galileo Galilei discovered the physical laws of falling bodies by dropping cannonballs from the Tower of Pisa. While setting up governing equations excels at capturing human knowledge within the system formulation, it is also very research-intensive and limited by expert knowledge. On the other hand, the ever-increasing availability of computing power and sensor technologies has spurred the development of new modes of modeling. This includes *data-driven* approaches whereby a model is directly "crafted" from (large-scale) time series data.

Nowadays, "machine learning" is often used as an overarching term to describe computerized system estimation to enable the analysis of systems for which rich observational data are available but principle terms are lacking. Often for these systems, the options to perform classical experimentation are also limited [Runge et al., 2019]. For example, in a "smart city" where a network of sensors measure the traffic flow, a data-driven model can predict the future traffic based solely on the historic observations [Lehmberg et al., 2021].

A drawback of data-driven modeling, however, is that model structures tend to become increasingly complex in the attempt to make more accurate predictions. Consequently, models are no longer simple enough for verbal interpretation, opposing von Neumann's requirement of a model. Or as Saltelli et al. [2020, p. 483] state, *"complexity can be the enemy of relevance"*. While a practitioner may be satisfied with a "black box"

model that can accurately forecast a system, a scientist may also wish to understand the information encoded in the model's structure that makes it an accurate model. Ideally, an *explainable* model provides descriptions to understand the model and interpret its operations [Gilpin et al., 2018]. This increased understanding helps to identify the situations for which the model assumptions are valid and to understand the uncertainties associated with the model. These aspects of computerized mathematical modeling have received much attention during the SARS-CoV-2 pandemic, especially if models predicting epidemic developments are integrated into political decisions [Saltelli et al., 2020]. However, there are no clear guidelines for what makes a model "interpretable". Gilpin et al. [2018, p. 81] state that *"for a system to be interpretable, it must produce simple enough descriptions for a person to understand using vocabulary that is meaningful to the user"*. From a mathematical perspective, this is a challenging task: Systems within a data-driven modeling context almost always show (unknown) nonlinear dynamics, but only linear systems have a well-founded mathematical theory. The situation becomes even more challenging when using multivariate, large-scale and noise-corrupted observational data.

1.1 Motivation and research questions

The goal of my thesis is to estimate dynamical systems from empirical time series data. I pursue an approach that allows researchers to predict and better understand dynamical systems. In my thesis, this includes systems from pedestrian traffic for which typically no governing equations are available.

My modeling approach centers around three concepts: time delay embedding, the Laplace-Beltrami operator and the Koopman operator. Table 1.1 provides the essence of each component in the context of my thesis. All three components were recently discovered to have deep theoretical connections to each other [Arbabi and Mezić, 2017; Das and Giannakis, 2019; Giannakis, 2019; Kamb et al., 2020]. These studies also demonstrate the analytical power of combining the components for estimating dynamical systems from data. Together the components relate to both the geometry and dynamics of the inferred system. Admittedly, these concepts can be difficult to grasp at first. However, I explain and build on the three components throughout my thesis, where I focus on software, numerical, and data-driven rather than theoretical treatment.

Operator-informed model approach

An operator-informed approach complements other modeling approaches from statistics and "classical" machine learning. The promise is to mitigate some of the common limitations of other methods and to provide a new perspective on the system representation that is easier to access.

I seek to combine the three components of Table 1.1 into a single operator-based model architecture in which each component fulfills a separate task. My particular interest is to explore the capabilities of this operator-informed approach within a "scientific machine learning" setting. This means that the modeling approach is guided by

Table 1.1: The main mathematical components and the associated numerical tasks used in my thesis, with a key reference for each component.

Component	Task	Reference
1 time delay embedding	state space reconstruction from partial observations	Deyle and Sugihara [2011]
2 Laplace-Beltrami operator	extract geometrically aligned state coordinates	Coifman and Lafon [2006b]
3 Koopman operator	extract linear dynamics in suitable state representation	Williams et al. [2015]

an underlying mathematical theory, here the operator theory. This aspect is interesting for data-driven analysis to increase scientific understanding of safety-relevant applications, such as the movement of crowds [Coveney et al., 2016]. A model forms a prediction based on intrinsic coordinates that connect to the theory and give insight into the identified system. The entire approach that I follow is data-driven and equation-free. Therefore, only minimal *a priori* system knowledge is required and the method is easy to transfer to new data applications.

Before I give an overview of the three components outlined in Table 1.1, I provide an intuitive example of an *operator*. Informally, an operator "manipulates" a function to obtain a new function. In other words, an operator is a "function of functions" [Levy, 2006]. Here I only consider the common case of *linear operators*. This means the operator map can be expressed similar to a matrix-vector product: $\mathcal{A}f(x) = f'(x)$. One example is a differential operator which is commonly written in Leibniz's notation, $\mathcal{A}f(x) = \frac{d}{dx}f(x)$. Applying the operator to a function of the monomial basis, $f(x) = x^n$, maps (linearly) to its derivative, $f'(x) = nx^{n-1}$.

Operator theory is a vast research field that studies linear operators as part of functional analysis. While the theoretical advancements are profoundly abstract and typically only accessible to mathematicians, this research field is also shown to be of high value for application-oriented problem settings [Coifman and Lafon, 2006b; Williams et al., 2015]. In this thesis, I focus on the value of operator theory in a data-driven context. For this, an intuitive understanding of an operator is usually sufficient and many concepts transfer to standard linear algebra. In the example above where $\mathcal{A}$ captures the derivative of the monomial basis, the *explicit* linear map can only be stored up to a finite degree. For capturing the *entire* map we could think of the true operator as being an infinite-dimensional matrix. The formalism of linear operators is also suitable to describe other and more diverse phenomena. However, unlike the "standard derivative map", the operator's structure is unknown and "hidden in data". In my thesis, the goal is to use operator theory as a way to (1) extract and describe the geometry and dynamics of observational time series data and (2) analyze the structure of the finite matrix (approximating the operator) in order to gain insight into the data-generating system.

In the scope of dynamical systems theory, Bernhard O. Koopman [1931] introduced an operator-based perspective on a dynamical system. For every canonical (nonlinear) dynamical system, there is a linear operator that describes the exact same dynamics. This means that from a Koopman perspective every dynamical system has a (typically unknown) operator that shifts functions linearly forward in time. Intuitively, this means we turn (nonlinear) observational time series data into a new time series with (approximately) linear dynamics. In fact, this is a popular idea in methods that are based on computational learning theory, whereby the original data is projected to a *feature space* in which the problem becomes easier to solve [Bishop, 2006]. A prominent example is the Support Vector Regression, in which nonlinear (static) observations are mapped to a (high-dimensional) feature space representation in which linear regression is applied [Drucker et al., 1997; Müller et al., 1997]. The Koopman operator methodology is similar in that it provides a theoretical framework that translates between time series states in a measurement and feature space (in both directions). Ultimately, given a suitable state representation, we can compute the Koopman matrix (approximating the operator) and reconstruction map with standard linear regression [Williams et al., 2015]. The major advantage is that capturing linear dynamics in a feature space is much easier to handle than a model that aims to capture nonlinear state dynamics directly. In this way, Koopman operator-based methods can be competitive against neural network-based models [Eivazi et al., 2021; Lange et al., 2021].

An important question, however, remains: How do we find a suitable state representation to approximate the Koopman operator? Addressing this question is a challenging task because it depends on several often unknown system factors [Williams et al., 2015]. Within the framework, this question is therefore left open and needs to be addressed for the concrete system. In my thesis, I follow a *geometric* approach, comprising the other two main components in Table 1.1 — time delay embedding and the Laplace-Beltrami operator. Both components embed into an (unsupervised) "representation learning" task to augment and extract hidden explanatory data features from the measured time series. In particular, the new states have high analytic value to describe and also model time series data [Berry et al., 2013; Dietrich et al., 2016; Giannakis and Majda, 2013]. Importantly, they provide a generic data transformation that is able to linearize the state dynamics and is, therefore, a great candidate within the Koopman operator framework [Arbabi and Mezić, 2017; Giannakis, 2019]. Time delay embedding is an established procedure in time series modeling and goes back to Takens [1981] theorem. The embedding can be seen as a *temporal* feature extraction because it can reconstruct essential dynamic information from partial system observations. The Laplace-Beltrami operator is a second-order differential operator and is a generalized form of the common Laplace operator (divergence of gradient, commonly denoted as $\Delta f(x) = \nabla \cdot \nabla f(x)$). The operator is of high interest because it encodes valuable and informative factors that relate to the geometry and data-generating process [Berry and Sauer, 2016; Levy, 2006]. For data-driven settings, there is a popular class of kernel-based methods that can approximate the Laplace-Beltrami operator from high-dimensional and unstructured data [Coifman and Lafon, 2006b]. The kernel describes the point relations in the data and is

the basis to extract the new geometric aligned data coordinates. As a secondary objective of my thesis I also contribute a novel algorithm that aims to accelerate computing a sparse kernel matrix. The algorithm is implemented in a separate software *rdist*.

The core advantage of describing a model architecture in terms of linear operators, is that the linear algebra machinery can be applied in a straightforward fashion. In particular, the operator's spectral components as the eigenvalues and eigenvectors (translating to eigenfunctions in operator theory) describe essential model characteristics. In my setting, this includes a description of the intrinsic geometry of spatio-temporal data [Berry et al., 2013] or the model's prediction stability. In light of these advantages, the Koopman operator methodology is frequently promoted as the main candidate for estimating, describing and controlling dynamical systems based on time series data [Budišić et al., 2012; Mauroy et al., 2020; Mezić, 2020; Surana, 2020].

Lack of scientific software

As already stated, my aim is to unify the three components in Table 1.1 into a single model architecture to reflect the underlying theory that connects these components. All three components come from diverse lines of research (e.g. geometry versus dynamics). Connecting them into a single data-driven model to estimate a dynamical system therefore requires transferring the components to a machine learning perspective.

Making use of operator theory in a data-driven setting means adopting numerical frameworks that are able to approximate the operators from data. Widely-used frameworks are the Diffusion Maps [Coifman and Lafon, 2006b] for the Laplace-Beltrami operator and the Extended Dynamic Mode Decomposition [Williams et al., 2015] for the Koopman operator. A major obstacle for modeling on concrete datasets, however, is a lack of available software solutions that fully reflect these frameworks. Partial software solutions exist, but are incompatible with each other and lack essential features needed to be connected in a single data processing pipeline. In my view, this absence of versatile, extensible and openly available software within a data-driven context of operator theory is a major shortcoming of the methodology as a whole.

A computational scientist is typically both a developer and user of scientific software. This makes software a central element in modern computerized research [Goble, 2014; Hannay et al., 2009]. In a data-driven modeling workflow, the building process and final model essentially stem from data and the source code. However, despite its importance for reproducibility and its value to enable new research, contributions to scientific software are considered secondary in the research community [Anzt et al., 2020; Hafer and Kirkpatrick, 2009]. This often leads to neglected software management and poor financing of software projects [Nowogrodzki, 2019]. As a result, the true potential of methodological innovations — here operator-informed machine learning — often remain hidden in the scientific literature [Rice and Boisvert, 1996]. While there is ongoing research to incrementally improve on numerical frameworks, this also increases the complexity and hence raises the barrier for newcomers and modelers to actually make use of the methodology. Source code that produces published results is

typically written in a "once-off" fashion and, moreover, often not openly shared [Goble, 2014; Sonnenburg et al., 2007].

Transferring methodological research to a problem-solving software is a nontrivial task, that requires developing skills and expertise within the application field, computer science and numerics [Rice and Boisvert, 1996; Wiese et al., 2020]. On the other hand, openly available software that unifies and transfers algorithmic innovations can tremendously accelerate a methodological approach as a whole. A good example is the neural network community where many scientists gather around large software projects — mainly *TensorFlow* [Abadi et al., 2016] or *PyTorch* [Paszke et al., 2019].

The promising operator-informed approach, as well as the current software gap, lead to the two central research questions of my thesis:

Research questions

1. How can the operator-informed modeling approach for system identification — comprising the Laplace-Beltrami operator and Koopman operator — be translated into a machine learning perspective?
2. Can the setting from (1) provide insight into the model and (hidden) dynamical system by making use of the operators that store geometric and dynamical information?

Regarding the **first question**, research within the operator-informed methodologies is highly active [Budišić et al., 2012; Mauroy et al., 2020]. Common research goals are to (incrementally) improve the quality of the approximation of the operators from data or to extend the methods to new problem sets. To answer my first research question I transfer widely-adopted numerical frameworks to the broader picture of a machine learning context. Furthermore, I address many aspects that are typically not discussed within the literature of operator-based research, such as numerical robustness, algorithmic efficiency, data structures, interface design, parameter optimization or dealing with missing data. Rather than being content with setting up a software solution that "just works", I aim to follow good practices of software engineering and management. Each component of the software should be usable on its own but also within a larger model architecture. While I rely on established software concepts of machine learning for static data, I also extend these to data with temporal context if necessary. Ultimately, a software solution that unifies the diverse concepts in Table 1.1, supports algorithmic experimentation and enables new research is the answer to the first research question of my thesis.

For the **second question** I apply the operator theory and approximation methods within the software solution to analyze three concrete scenarios of time series data. In this operator-informed setting, I want to promote and discover the possibilities of the innovative methods within a "scientific machine learning" context. I explore the operator-informed approach to not only perform accurate predictions, but also to extract geometric and dynamic coordinates from time series data. I account for a wide

range of challenges within the three scenarios, such as diverse temporal patterns, high dimensional data, noise corrupted measurements and intervals of missing data. The model approach and software should be robust to these factors. The urgent need for software solutions is also highlighted by the fact that several new software projects with a similar objective were initiated since I commenced my thesis [de Silva et al., 2020; Hoffmann et al., 2021; Martensen and Rackauckas, 2021].

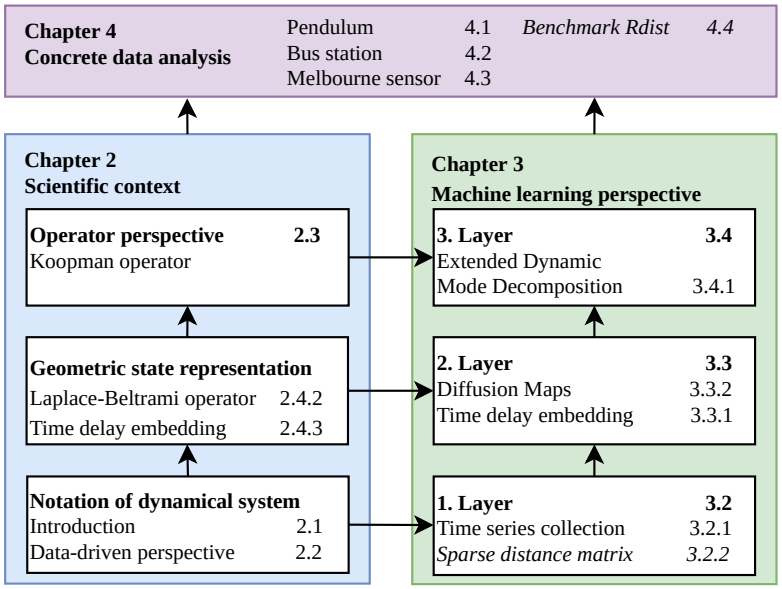


Figure 1.1: Visualization of the main methodological components of my thesis. All numbers refer to sections within the thesis. Chapter 2 focuses on the methodological background that is required for the machine learning perspective of Chapter 3. Chapter 4 then includes concrete data analysis, which requires both of the preceding chapters to interpret the model's components and software solution. Sections in *italics* denote an additional contribution that follows a secondary objective to accelerate kernel-based methods (not covered by the main research questions).

1.2 Structure of the thesis

I divide my thesis into three main chapters, which are visualized in Fig. 1.1. Note that in Chapter 2, I follow a top-to-bottom approach. I first describe the operator perspective for a dynamical system before I highlight the need for a suitable state representation. Conversely, I follow a bottom-to-top approach in Chapter 3, because this successively increases the complexity of the software methods involved.

Chapter 2 begins with a general notation of dynamical systems as well as an overview of common data-driven approaches to extract a model from time series. Here I mostly focus on a data-driven and discretized perspective that becomes relevant to address the first research question. In Section 2.3, I introduce the Koopman operator and its nu-

merical treatment as the main modeling approach for dynamical systems in the thesis. In Section 2.4, I introduce a composed data transformation, which comprises the time delay embedding and the eigenfunctions of the Laplace-Beltrami operator. This also includes research links back to the Koopman operator. In Section 2.5, I introduce the software perspective on data-driven modeling of dynamical systems. Here I describe the "system identification loop" that mandates a flexible software design and give an overview of relevant available scientific software within the Python programming language.

Chapter 3 addresses the first research question and marks the first contribution of my thesis. I describe *datafold* as a sustainable software in which I transfer the mathematical methods covered in the second chapter from a machine learning perspective. The software promotes operator-based modeling of dynamical systems and covers the entire machine learning pipeline in a hierarchical structure that mirrors the system identification loop. This also includes additional aspects of software design and data-driven modeling that are not addressed within the operator literature. Section 3.3 describes a data structure for time series to enable important model generalizations. Here I also contribute a new fundamental algorithm to compute a sparse distance matrix, with the objective to accelerate the computation of a kernel matrix, as a basis to extract geometric coordinates. In Sections 3.4 and 3.5, I transfer the numerical frameworks Diffusion Maps (Laplace-Beltrami operator) and Extended Dynamic Mode Decomposition (Koopman operator) to approximate the respective operators from finite data.

Chapter 4 contains my second thesis contribution. This includes the concrete analysis of simulated and real-world data with the operator-informed setting. The modeling and analysis would not be possible without the software from Chapter 3. In the first scenario, I analyze a simple pendulum system as an "academic example". This allows me to both validate the main setting against available equations and to highlight the functionality of *datafold*. The following two data scenarios "Bus station" (Section 4.2) and "Melbourne sensor data" (Section 4.3) come from the field of crowd dynamics. These traffic systems exhibit complex interactions for which no governing equations are available. For the "Bus station" scenario I exemplify the model approach in an uncertainty quantification analysis of a microscopic pedestrian simulator. The "Melbourne sensor data" represents the most challenging analysis. Here I use multi-variate and real-world sensor data that are noise-corrupted and time series come with intervals of missing data. I finish the chapter with a separate benchmark analysis on static data (Section 4.4) as a result of my algorithmic contribution of accelerating the sparse neighborhood computation from the previous chapter.

Finally, in Chapter 5 I reflect on the contributions and value of my thesis results for advancing methods for the data-driven analysis of dynamical systems. By contributing to an active research field, I also highlight promising directions for future software development and research. All software that I developed for the thesis and scripts and data for the analysis are included in the Supplementary Material.

2 Scientific context

In this first chapter, I describe the scientific context of my thesis. I transfer the central mathematical concepts outlined in this chapter into a software solution (Chapter 3) and use them to analyze concrete data scenarios (Chapter 4). The modeling approach brings together different mathematical theories. I intend to highlight the essence of each of the concepts from a data-driven perspective. I leave the mathematically rigorous explanations to the linked literature.

In Section 2.1, I briefly introduce dynamical systems as a broad research field. Starting from Section 2.2, I then focus on a data-driven modeling perspective and in Section 2.3 highlight the Koopman operator as the main framework in my thesis to estimate a dynamical system. In Section 2.4, I highlight the Laplace-Beltrami operator and time delay embedding within a geometric perspective of time series data. Finally, Section 2.5 describes the "system identification loop" as a general data-driven workflow that mandates high flexibility in scientific software. I also give an overview of software packages in Python as a popular programming language for data-driven tasks.

2.1 What is a dynamical system?

Dynamical systems are a mathematical framework to describe, analyze and predict processes that evolve with time. The discipline has a rich history that includes many turning points in how science understands time-dependent processes in nature. While *time* itself could be interpreted as a system variable along with spatial coordinates, because of its peculiar properties and the wide range of temporal-specific phenomena it is typically treated separately. In this section, I briefly describe some important concepts of dynamical systems. For a comprehensive introduction I refer to Layek [2015] and Strogatz [2015].

A dynamical system consists of the two components of a *state* and an *evolution rule* to describe the change of states over time. Established in the 17th century by Isaac Newton and Gottfried Leibniz, the mathematical formalism for describing time-dependent systems are ordinary differential equations (ODEs). The system is represented with a smooth (Lipschitz continuous) function f that describes an infinitesimal change in time

$$\frac{d}{dt}\mathbf{x}(t) = f(\mathbf{x}(t)), \tag{2.1}$$

acting as the evolution rule on the system's state in a numeric column vector of size N, $\mathbf{x} \in \mathbb{R}^N$. A state contains all system-relevant quantities, which can have different physical units and meaning (e.g. pressure and velocity). But it is assumed that all

quantities originate from the same underlying system to be modeled. Time itself is a real-valued scalar value and typically positive, $t \in \mathbb{R}^+$. Note that the evolution rule itself could also change with time — in which case the description of state evolution has time as an additional argument $f(\mathbf{x}, t)$ — but for simplicity, I only consider *autonomous* systems.

The differential form in Eq. 2.1 is still omnipresent in mathematical modeling — from describing classic physical laws to data-driven approaches. The function $f(\mathbf{x})$ describes a "vector field", because it assigns to each state $\mathbf{x}$ a vector $^{d}/_{dt}\ \mathbf{x}$ which indicates the direction of change. An example of a vector field is given in the left plot of Fig. 2.1.

To perform future state predictions of a system, we can evaluate the *flow*, denoted as $F_{\Delta t}$, which is obtained by integrating along the vector field with

$$\mathbf{x}(t_2) = F_{\Delta t}(\mathbf{x}(t_1)) = \mathbf{x}(t_1) + \int_{t_1}^{t_2} f(\mathbf{x}(\tau))d\tau. \tag{2.2}$$

Instead of describing the *change* of a state, the flow $F_{\Delta t} : \mathbb{R}^N \to \mathbb{R}^N$ maps an (initial) state $\mathbf{x}(t_1)$ to a future state $\mathbf{x}(t_2)$ with time increment $\Delta t = t_2 - t_1$. The flow map satisfies the semi-group property such that $F_t(F_s(\mathbf{x})) = F_{t+s}(\mathbf{x})$. Evaluating the flow at multiple discrete future times $t \in \{t_1, t_2, t_3, \ldots\}$ (with $t_{j+1} > t_j$) from an initial condition at t_1 then describes a solution trajectory (or orbit for a trajectory that is continuously evaluated). The left plot in Fig. 2.1 shows two example trajectories with different initial states within the vector field, in which the trajectory follows the direction of the local vector. The right graphs of the figure show how the two state coordinates change over time.

Once a flow representation of Eq. 2.2 is available, it can then be used to generate solutions of prediction tasks, based on a given initial state $\mathbf{x}(t_1)$. However, with the exception of linear and a few nonlinear systems, it is not possible to obtain a closed-form solution of the flow. Instead, one has to make use of numerical integration schemes, such as the Euler or Runge-Kutta scheme.

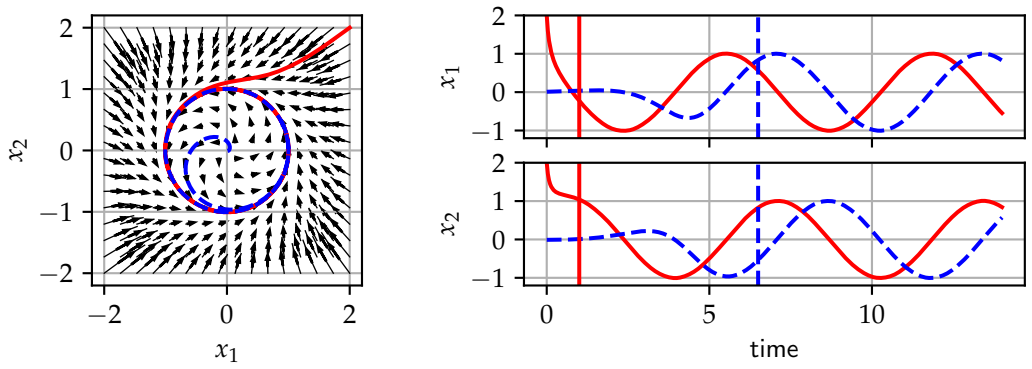


Figure 2.1: Example of a Hopf differential system. The dynamics are given with, $^d/_{dt}\ x_1 = -x_2 + x_1(1 - x_1^2 - x_2^2)$ and $^d/_{dt}\ x_2 = x_1 + x_2(1 - x_1^2 - x_2^2)$ and is taken from Bollt [2021, Example 2]. Left: The vector field in the plane. The length of a vector correlates with the rate of change at the respective state. The two example trajectories have an initial condition $\mathbf{x}_1 = [2, 2]$ (red) and $\mathbf{x}_1 = [0.1, 0.1]$ (blue) respectively. Right: The solution trajectories plotted over time. Both trajectories converge to the stable limit cycle as an attractor of the system, where the vertical lines mark the time at which the respective trajectory is close to the attractor.

An example of a "simple system" that has been proven to have no analytical solution is the notorious three-body problem, describing the orbits of three planets under gravitational force. Based on this problem, in 1890, Henri Poincaré developed a new perspective that marks the beginning of the scientific field of "dynamical systems" [Layek, 2015]. While the mathematical system representation of Eq. 2.1 – 2.2 remains the same, the novelty in Poincaré's perspective is to collectively view all possible system states in a single *geometrical* object, which is referred to as the *state space* of the system. Instead of quantitative future state evaluations, which require an initial state, the geometric analysis is suitable to identify qualitative system characteristics [Layek, 2015]. For example, the "Poincaré map" can evaluate a system's stability or periodicity by recording intersections of a trajectory with a hyperplane in the state space. In Fig. 2.1 the state intersections with a line at $x_2 = 0$ (corresponding to a low-order hyperplane) would reveal that after a transition phase the system is periodic and stable.

For systems with a low state dimension (small N) the state space geometry can be curves or planes as shown in Fig. 2.1. In a more general notion, the geometry is described as smooth and curved surfaces in higher-dimensional spaces (leaving out more complicated structures such as fractals here). These geometric objects are referred to as *manifolds* and I give a brief introduction to these geometric objects in a data-driven context in Section 2.4.

The state space contains all possible solution trajectories $\mathbf{x}(t)$ that can be generated with Eq. 2.2, because any point of the state space can serve as an initial state; Fig. 2.1 includes two example trajectories. The state space itself can also be subdivided into geometric objects, describing different characteristic parts of the dynamical system. For example, *attractors* describe a part of the state space to which many trajectories converge

after a transient phase. Attractors can be a single point where all points converge to, but also generalize to a manifold. In the example of Fig. 2.1 the system has a circle as an attractor, which is referred to as a "limit cycle" [Layek, 2015, Sec. 5.5].

The analysis of a dynamical system includes classifications about its long term behavior. A major interest are *ergodic* systems in which every state of the system returns to all other states of the state space in the long term ($t \to \infty$). In the example of Fig. 2.1, only the states of the limit cycle and the (unstable) point at the origin are ergodic. All other states are transient and the system never returns to these states. Ergodicity comes with many suitable properties and is an often-stated assumption for the analysis of dynamical systems.

Moreover, dynamical systems can also be described on multiple scales, ranging from fast to slow dynamics. Often only *slow manifolds* are considered which describe a reduced dynamical system that only contains the principal dynamics [Dsilva et al., 2016]. This is because in terms of the slow dynamics the system reduces to the essential description that is relevant for long term predictions [Gear et al., 2005]. On the other hand, fast dynamics are more corrupted by noise and measurement errors.

Overall, the geometric notion in dynamical systems as introduced by Henri Poincaré has advanced the analysis of nonlinear dynamics and profoundly changed the understanding of many natural phenomena. One landmark research of Edward N. Lorenz [1963] could show that even seemingly simple deterministic differential systems exist, that have solutions that are not predictable within bounds for larger time horizons. What is also known as the "butterfly effect" could explain why weather forecasts fail for larger time horizons. Tiny differences in a state in the order of machine precision, which can be introduced by measurement errors or numerical integration schemes, can lead to fundamentally different long term solutions. Such systems complicate the state space geometry to fractals and are characterized as *chaotic*. Another important contribution was made by Takens [1981], who provides a theory to reconstruct a qualitative copy (diffeomorphic, in mathematical terms) of an attractor manifold, if only partial observations are available. This is particularly important for settings in which measurements contain physical (spatial) quantities that have insufficient temporal information.

In recent decades, the perspective and modeling of dynamical systems have shifted to a new data-driven era. Unlike solving differential systems with numerical algorithms, such as the Navier-Stokes equations in the field of computational fluid dynamics, the goal is to *identify* complex dynamical systems based on empirical data, because an explicit state evolution (the differential f or flow $F_{\Delta t}$) are hard or impossible to obtain. This branch of dynamical systems connects to the general trend in science to data-driven and equation-free modeling to approximate systems through observational data [Brunton and Kutz, 2019].

Linear dynamical systems

In the theoretical foundation of dynamical systems, the best developed systems are ones with *linear* dynamics. In contrast, processes observed in nature have almost always nonlinear state interactions. This creates a large gap between practice and math-

ematical foundation, because directly applying the theoretical rich formalism to real-world observations usually fails dramatically. A common procedure to still profit from a deeper understanding of linear dynamical systems is to approximate the nonlinear system in a smaller regions of interest, such as around an equilibrium state [Brunton and Kutz, 2019].

Here I continue to highlight the describe linear dynamical systems and how these profit from the linear algebra machinery. The main reason is that these basic concepts prepare the operator view on dynamical systems. In Section 2.3, I show that the Koopman operator provides a promising framework that has the potential to close the gap between nonlinear systems and linear system theory. Many of the statements highlighted here have their equivalent counterpart in the operator view.

A vector field of a linear dynamical system has the form

$$\frac{d}{dt}\mathbf{x}(t) = \mathcal{A} \cdot \mathbf{x}(t), \tag{2.3}$$

where $\mathcal{A} \in \mathbb{R}^{N\times N}$ is a time-invariant system matrix, acting on a column-vector state $\mathbf{x} \in \mathbb{R}^N$. Eq. 2.3 corresponds to a homogeneous linear differential equation, which has a closed-form solution for the flow

$$\mathbf{x}(t) = \exp(\mathcal{A}t)\mathbf{x}(t_1), \tag{2.4}$$

where $\exp(.)$ corresponds to the matrix exponential and $\mathbf{x}(t_1)$ the initial condition (commonly $t_1 = 0$). Since a unique solution exists for any initial state, it follows that the trajectories never intersect on the state space. With a constant sampling rate, $t_{j+1} = t_1 + j\cdot\Delta t$, the flow describes a discrete map of the continuous dynamical system

$$\mathbf{x}_{j+1} = \overbrace{\exp(\mathcal{A}\cdot\Delta t)}^{A_{\Delta t}}\mathbf{x}_j = A_{\Delta t}^j\mathbf{x}_1, \tag{2.5}$$

for $j = (1, 2\ldots)$. The system matrix for the continuous system representation in Eq. 2.3 induces a family of discrete maps with sampling intervals $\Delta t > 0$. It is therefore also referred to as the *generator* of the matrices $A_{\Delta t}$.

The discrete flow representation is often more suitable for data-driven modeling, because it matches the discrete measurement events in the available time series data. Approximating a system from data with a linear model form as per Eq. 2.5, requires finding the system matrix $A_{\Delta t}$, which can be achieved by solving a linear regression problem $A_{\Delta t}\mathbf{x}_j \approx \mathbf{x}_{j+1}$ of available snapshot pairs $(\mathbf{x}_j, \mathbf{x}_{j+1})$.

Alternatively, the generator matrix $\mathcal{A}$ itself can be approximated, which requires pairs of state and corresponding derivative, $(\mathbf{x}_j, {}^{d}/_{dt}\,\mathbf{x}_j)$. However, the derivatives are often not available or can only be approximated with finite difference schemes. Because of the clear correspondence between $A_{\Delta t}$ and $\mathcal{A}$ in Eq. 2.5 the generator can also be obtained with the matrix-logarithm $\mathcal{A} = {}^{\log A_{\Delta t}}/_{\Delta t}$. However, this operation may not be well-defined; for details see Dietrich et al. [2020]; Mauroy and Goncalves [2020]. To avoid

such issues, the relation can also be expressed through an Euler discretization of the system in Eq. 2.4

$$\mathbf{x}_{j+1} \approx \overbrace{(I + \Delta t \mathcal{A})}^{A_{\Delta t}} \mathbf{x}_j, \tag{2.6}$$

where $I \in \mathbb{R}^{N \times N}$ is the identity matrix. From this follows that $\mathcal{A}\mathbf{x} = \lim_{\Delta t \to 0} {}^{(A_{\Delta t}\mathbf{x} - \mathbf{x})}/_{\Delta t}$ as a common representation of the generator matrix found in literature [e.g. Dietrich et al., 2020; Klus et al., 2020; Otto and Rowley, 2021].

A major advantage of linear dynamical systems over nonlinear ones is that the representation can be easily transformed into system-intrinsic components. In particular, the system matrix can be decomposed into spectral components, which serve as spatio-temporal "building blocks" that are easier to comprehend.

For simplicity, we assume that the system matrix is diagonalizable $A_{\Delta t} = \Phi \Lambda \Phi^{-1}$, where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_N)$ is a matrix with eigenvalues on the diagonal, Φ contains the right eigenvectors in the matrix columns and Φ^{-1} the left eigenvectors in the rows (by convention). All three spectral components are considered complex-valued and of size $\mathbb{C}^{[N \times N]}$. Following up on the discrete dynamical system in Eq. 2.5, the spectral representation becomes

$$\mathbf{x}_{j+1} = \Phi \Lambda^j \Phi^{-1} \mathbf{x}_1 \tag{2.7}$$

$$= \Phi \Lambda^j \mathbf{b}_1 = \sum_{n=1}^{N} \phi_n \lambda_n^j [\mathbf{b}_1]_n. \tag{2.8}$$

In the last statement $\phi_n \in \Phi$ is the n-th column vector and the initial state is spectrally aligned by the left eigenvectors, $\mathbf{b}_1 = \Phi^{-1}\mathbf{x}_1$ ($[\mathbf{b}_1]_n$ denotes the n-th element of the vector $\mathbf{b}_1$). The spectral form of the discrete system again relates to the underlying respective continuous system. While the left and right eigenvectors remain the same for both systems, the eigenvalues of the differential form are $\mu_n = \log(\lambda_n)$, which follows from the system relationship in Eq. 2.5.

The main advantage of the spectral representation is that the only time-dependent quantities of the system are the eigenvalues; both the eigenvectors and initial condition remain constant throughout a prediction. This gives valuable insight into the long term behavior of the system. Table 2.1 lists the stability criteria of the respective system type, which can be directly read from the eigenvalues. For a discrete map the corresponding eigenvalue's magnitude, $|\lambda_n|$, indicates how the product in Eq. 2.8 evolves over time. The overall system is considered asymptotically stable if *all* eigenvalues, $n = (1, \ldots, N)$, are stable.

Table 2.1: Stability criteria for a linear dynamical system in discrete or continuous form.

Continuous (Eq. 2.3)	**Discrete** (Eq. 2.5)	**Stability**
$\Re(\mu_n) = 0$	$\lvert\lambda_n\rvert = 1$	stable, long term
$\Re(\mu_n) < 0$	$\lvert\lambda_n\rvert < 1$	stable, converging to zero
$\Re(\mu_n) > 0$	$\lvert\lambda_n\rvert > 1$	unstable, exponential growth

As highlighted before, using a linear system form is usually only feasible to analyze local regions of an underlying nonlinear system. In the next section, I highlight data-driven approaches that aim to estimate global (nonlinear) representations of a dynamical system.

2.2 Data-driven estimation of dynamical systems

Based on the notion of a dynamical system introduced in the previous section, I now follow up on a data-driven perspective. Data-driven modeling bypasses the workflow of more traditional approaches: Instead of integrating knowledge of first-principles into explicit equations (or update rules within larger simulation codes), the model is directly "crafted" from empirical observation data [Tangirala, 2018]. With a suitable model structure, the model's quality is more limited by the quality of the data or required computational resources and less by the complexity of the actual system itself.

Many data-driven modeling approaches only require minimal prior knowledge of a system, which makes it especially useful when the underlying system is too difficult or complex to obtain governing equations. But even for systems for which there are explicit equations available, data-driven models are increasingly often integrated [Brunton and Kutz, 2019]. This is because measurement data can include a wealth of information about the system, that may not be covered by first-principle terms, potentially leading to better performance measures. Besides using real-world measurements (e.g. from sensors) data-driven approaches also complement the analysis of scientific simulations, often producing large amounts of data [Dietrich et al., 2018].

In the next section, I first describe the term "system identification" as the process of building models of dynamical systems from the empiric relations in time series data. Section 2.2.2 highlights challenges that are introduced in system identification and Section 2.2.3 gives an overview of model approaches from different research directions.

2.2.1 System identification

The discovery process to describe the relationship of observational data (input to output) has its origins in probability theory and statistics. In the era of "big data" and increased computational resources, new types of algorithms have evolved into the field of *machine learning* along with a new terminology [Murphy, 2012]. While the overall goal of finding a suitable model based on empirical data remains, in contrast to its sta-

tistical heritage machine learning algorithms often better scale with the amount of data, both in the number of samples and dimension.

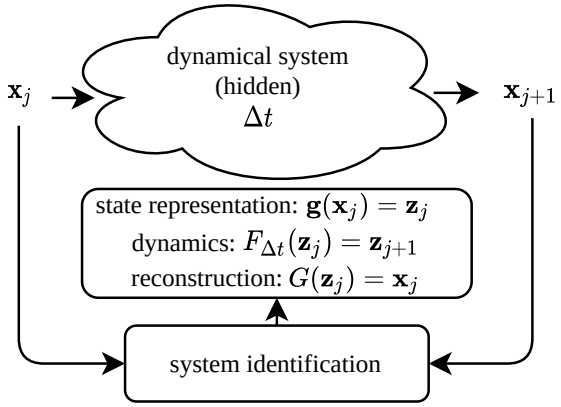


Figure 2.2: Schematic illustration of system identification to build a model from time series data as snapshot pairs $(\mathbf{x}_j, \mathbf{x}_{j+1})$ of input and output of the dynamical system. The two main tasks are (1) find a suitable state representation $\mathbf{g}(\mathbf{x}) = \mathbf{z}$ and (2) model the dynamics in this state $F_{\Delta t}(\mathbf{z})$ and (3) reconstruct the observational data $G(\mathbf{z})$ (cf. Eq. 2.10 – 2.11). Adapted from Tangirala [2018, Fig. 1.1.].

Machine learning in a "traditional sense" often disregards a temporal relation in data and views it as static. However, the broad categories of machine learning [Murphy, 2012]

- *supervised learning* — model the functional relation $\mathbf{h}(\mathbf{x}) = \mathbf{y}$ between input and target $\{\mathbf{x}_j, \mathbf{y}_j\}_{j=1}^{J}$
- *unsupervised learning* — extract a suitable data representation $\mathbf{g}(\mathbf{x})$ from the (unlabeled) data itself $\{\mathbf{x}_j\}_{j=1}^{J}$.

transfer to the modeling of dynamical systems. The goal is to capture the dynamics through either the vector field f (Eq. 2.1) or flow $F_{\Delta t}$ (Eq. 2.2). As a distinction to static data, the modeling procedure for dynamical processes is referred to as *system identification*, which as described by Tangirala [2018, p. 2] "*concerned with the means and techniques for studying a process system through observed / experimental data, primarily for developing a suitable (mathematical) description of that system*". Fig. 2.2 highlights the input-output relations in the data that are the basis for the modeling. Since this temporal context is essential for system identification, the available data is represented as a time series

$$X = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_J], \tag{2.9}$$

where $X \in \mathbb{R}^{[N \times J]}$ is in a matrix form. The time series has N rows, corresponding to the spatial states at a given measurement time (also referred to as *snapshots*), and J

columns that describe the temporal evolution of the snapshots in discrete time steps with a constant time interval $\Delta t > 0$. Each (column) vector $\mathbf{x}_j$ has an associate *unique* time value $t_{j+1} = j \cdot \Delta t$ (with $t_1 = 0$). The time values impose an inherent order on the data such that $t_j < t_{j+1}$.

An important aspect in modeling is that the measurement data $\mathbf{x}$ — as a fundamental source in the modeling procedure — is often not suitable to describe the dynamical system. Possible reasons are more detailed in Section 2.2.2 below. The emerging and challenging task then becomes to first extract a suitable state representation from the available data [Pintelon and Schoukens, 2012]. This is summarized in the general state-space representation for nonlinear system identification [Nelles, 2020, Eq. 19.6a - 19.6b]:

$$\mathbf{z}_{j+1} = F_{\Delta t}(\mathbf{z}_j) + \varepsilon_j \tag{2.10}$$

$$\mathbf{x}_{j+1} = G(\mathbf{z}_{j+1}) + \nu_{j+1}, \tag{2.11}$$

where $F_{\Delta t} : \mathbb{R}^P \to \mathbb{R}^P$ is again the flow (a similar system can be set up in terms of the vector field f). However, instead of acting on the measurement states $\mathbf{x}$ directly, the flow acts on a new (intrinsic) state vector $\mathbf{z} \in \mathbb{R}^P$ that is suitable to describe the spatio-temporal system dynamics. The additional function $G : \mathbb{R}^P \to \mathbb{R}^N$ reconstructs the original measurements states from the intrinsic state. The terms ε_j and ν_j account for time-dependent system and measurement noise respectively.

The future state $\mathbf{z}_{j+1}$ in Eq. 2.10 is solely based on the current state $\mathbf{z}_j$. In engineering settings the flow $F_{\Delta t}(\mathbf{z})$ often also includes additional variables to capture exogenous forcing, such as from boundary conditions [Nelles, 2020; Pintelon and Schoukens, 2012]. However, in the context of my thesis I leave out the treatment of such additional terms. I instead focus on learning the (unforced) flow $F_{\Delta t} : \mathbb{R}^P \to \mathbb{R}^P$ and reconstruction map $G : \mathbb{R}^P \to \mathbb{R}^N$ (Section 2.3) as well as on extracting a latent state representation $\mathbf{g}(\mathbf{x}) = \mathbf{z}$ (Section 2.4). The later corresponds to an unsupervised task, whereas finding the two system functions is a supervised task.

While there is a common goal in system identification to approximate an observed dynamical system well, the applications of a model once it is available can differ. The following list is adapted from [Kutz et al., 2016a]:

- **Diagnostics**
  Given the initial motivation to make use of data-driven modeling is to identify systems of which governing equations are not available, it is a natural application to use a model to obtain a deeper understanding of the process [Berry et al., 2020; Brunton et al., 2016].

  However, this is often a challenging task, as often machine learning methods are black-boxes, compared to fully transparent first principle models. As a compromise, some methods allow incorporating *a priori* knowledge, such as physical understanding (e.g. conservation laws), structural constraints or symmetry [Karniadakis et al., 2021; Tangirala, 2018]. To decrease the complexity of high-dimensional states, a typical approach is to describe the system state in fewer and easier to comprehend coor-

dinates (i.e. $P \ll N$). As a result methods for dimension reduction, such as Singular Value Decomposition (SVD) become important [Tu et al., 2014].

- **Prediction**
  The recursive model description in Eq. 2.10 makes it straightforward to perform predictions over larger time horizons than simply the next state. A fundamental model requirement is that the model behaves "reasonably" in that it performs predictions that are covered by the example time series.

  Moreover, a successful model that can predict a dynamical system well can also be used to monitor a process. This is considered a special task of prediction, in which the goal is to identify unusual discrepancies between predicted and observed states [Zameni et al., 2019].

- **Control**
  In many engineering applications it is possible to manipulate the state of a system through additional input quantities in Eq. 2.10 – 2.11. It then becomes possible to change the characteristics of the system measurements over time, leading to the broad field of control theory [Brunton and Kutz, 2019; Mauroy et al., 2020].

The three tasks tend to increase in their difficulty. This means that finding a model to diagnose the system's characteristics is often easier than controlling it.

In the data analysis in Chapter 4, I focus on a model architecture that addresses the first two tasks. This means gaining insight into the dynamical system while also obtaining an accurate prediction model. Since I do not include forcing terms in the system formulation used in this thesis, I do not consider control tasks.

System identification is faced with specific challenges that are introduced when dealing with temporal data. The next section includes a list of challenges that can largely influence the choice of which modeling approach to use (an overview is given in Section 2.2.3).

2.2.2 Challenges in system identification

While the modeling efforts of a data-driven approach are often significantly reduced compared to explicit equation-driven modeling, the approach comes with its own set of challenges. Because observation data are a fundamental factor, a model can only be as good as the data quality. Moreover, compared to data-driven modeling with static data, there are key challenges and aspects that are introduced by the temporal aspect of dynamical systems and their observations in time series data. These are summarized in the non-exhaustive list.

- **Nonlinear dynamics**

  Most, if not all, processes in nature show nonlinear state evolution in the flow of a system. A successful data-driven model needs to capture this. However, in contrast

to static data, the dynamics can introduce a new set of often difficult to comprehend phenomena. These patterns could include transient, periodic, quasi-periodic to chaotic system behaviors [Brunton and Kutz, 2019]. There can be also system configurations (bifurcation) in which the system behavior fundamentally changes [Kevrekidis and Samaey, 2009]. In general, nonlinear dynamics can limit the predictability of a system, which is mathematically described by the Lyapunov exponent which can be associated with every dynamical system. Typically, when data-driven modeling is applied, however, the exact characteristics of the dynamics are unknown and need to be extracted from the available empirical data.

- **Recursive model evaluations**

  The identified flow of a system in Eq. 2.10 describes a recursive model, in which any predicted state $\mathbf{z}$ can be again fed back to the model's input [Beckers and Hirche, 2020]. This allows simulations over larger time horizons to be performed based on a single initial state. This contrasts to a typical single input and output model in time-independent regression problems. A challenge is that the recursive evaluation of the model prediction errors can quickly accumulate, especially if the underlying dynamics are highly nonlinear [Tangirala, 2018]. This can lead state predictions into regions that are not covered by the observation data, resulting in nonsense solutions. In prediction settings it is therefore an important task to evaluate both the forecasting stability of a model and a prediction horizon to which the model remains in required accuracy bounds.

- **Noisy observations**

  For many real-world datasets, the time series snapshots are corrupted by measurement errors and process noise. Part of the system identification is then to capture the deterministic underlying dynamics supported by data, of which there can be patterns on different scales [Berry et al., 2013]. If a model tends to also capture unpredictable noise terms it *overfits* the data. While this is also the case for static datasets, an important characteristic of temporal processes is that noise itself can be time dependent. It therefore violates the common assumption in regression methods of independent and identically distributed (i.i.d.) data.

- **Insufficient dynamical information**

  Typically, a measurement state $\mathbf{x}$ contains physical quantities of interest. However, if these quantities do not provide sufficient dynamical information, then approximating the flow in Eq. 2.10 based on data is not well-defined. For example, imagine a set of sensors producing a time series that describe the temperature over a domain. Because the physical quantity "temperature" does not include an instantaneous change in the form of a derivative or no trend information on larger time scales, the state dynamics are ill-defined. Increasing the number of temperature sensors only leads to better sampling of the spatial dimension and therefore does not solve the problem. Uncovering latent variables that are *dynamically* relevant is a major challenge in system identification [Brunton and Kutz, 2019; Dietrich et al., 2016]. Fortunately, in such

cases, the inherent temporal context in time series data is available to enrich a state. This can be achieved, for example, by estimating the derivative or applying state space reconstruction methods [Deyle and Sugihara, 2011; Takens, 1981].

- **Sampling distribution on state space**

  Data-driven models usually only operate well in regimes that are sufficiently covered by the data. A model that aims to provide a global system description can therefore vary in accuracy, depending on the sampling. Naturally, rare events occurring in a system are sparsely sampled. That such events are often poorly covered in data-driven models, is an inferior property compared to models that are based on governing equations [Tangirala, 2018].

  The sampling procedure is, therefore, an important part of data-driven modeling. While static datasets sample only spatial dimensions, time series data additionally includes a temporal dimension in the sampling procedure; corresponding to the rows and columns in Eq. 2.9. An important characteristic is the sampling rate Δt between samples. This is because it limits the dynamical patterns that can be extracted by a model, relating to the Nyquist–Shannon theorem in signal processing.

  System identification based on simulated data — for example, to build a surrogate model as in Dietrich et al. [2016] — is a privileged situation in that is possible to strategically sample the state space and select important state variables. In contrast, for real-world data there is often no access to adapt the characteristics of the sampling procedure. Measurements may be corrupted or not be reliable in that the observations are interrupted.

- **High-dimensional states**

  Many complex dynamical systems are described in terms of a high dimensional state to describe a spatial domain well (i.e. $N \gg 1$). Consequently, identifying a flow F from data is usually a multi-valued regression problem, with the same dimension in the input and output. While classical regression models of machine learning are designed to handle high-dimensional input, many standard methods often only support single-valued output. A naive approach is to construct a separate model for each state variable (i.e. N models). However, this increases the overall modeling efforts, model complexity and computational requirements.

- **Generalizing a model**

  In data-driven modeling it is vital to verify whether a model *generalizes* to situations that are new to the model [Ding et al., 2018]. That is, to empirically validate that the model performs well in situations that are not covered by the data used for the model construction [Bergmeir and Benítez, 2012]. There are established procedures, such as cross-validation, that allow estimating the generalization error [Bergmeir and Benítez, 2012]. However, the temporal order in the data prohibits common good practices such as taking random subsets of the data as this breaks important characteristics of the data [Bergmeir and Benítez, 2012]. Moreover, as highlighted above,

snapshots in time series data are often not i.i.d. and the error may then be biased by the specific data allocation.

2.2.3 Modeling approaches

In addition to the quality of underlying time series data, a crucial factor for system identification is *model selection* [Bergmeir and Benítez, 2012; Murphy, 2012]. In a first step, this means to select a suitable model structure, for example, a linear, polynomial or neural network type of model. Only in a second step, a suitable parametrization needs to be found [Murphy, 2012]. The model structure greatly influences the character of the model and an improper choice can lead to severely misleading conclusions and disappointing predictive performance [Åström and Eykhoff, 1970; Ding et al., 2018].

Technically, a model selection is understood as an optimization over the model and parameter space to find the best model among candidates. The optimization is performed by minimizing some cost function, which could be the prediction error over a specified time horizon. However, the selection of candidate models is also constrained by a myriad of additional factors, including available *a priori* knowledge of the system, the state dimension, its intended use, computational requirements or available software [Åström and Eykhoff, 1970; Nelles, 2020, p. 11]. A well-suited model has a structure that meets the requirements of the application. This means that the optimization does not have to be strict; a simple model should be favored over a complex one if the accuracy improvements are only marginal [Tangirala, 2018].

Broadly, different model structures can be classified in terms of the number of parameters [Åström and Eykhoff, 1970; Murphy, 2012]:

- **Parametric models** have a *fixed* number of parameters. The models include strong assumptions about the underlying data relation, which reduce the optimization problem to parameter estimation. If the specified model complexity does not match the system properties, this can result in a strongly biased model and large errors. An example of a parametric model, is a linear or polynomial model form because the number of coefficients is constant.

- **Non-parametric models** have a *variable* number of internal parameters. This means that this model type can "automatically" increase the number of internal parameters to better describe the data. The greater flexibility comes with higher computational demands and the models are prone to overfitting, whereby the model complexity is much larger than the one covered by the data (or system). Typically, the models contain *hyperparameters* which control the number model parameters [Murphy, 2012]. An example of a non-parametric model is a decision tree, which can "grow" indefinitely by adding more parameters to describe the decision branches.

In the common setting in which prior knowledge of principle dynamics is scarce, non-parametric are usually more suitable, because they are more generic compared to

parametric models. As a consequence, for complex and high-dimensional systems, they tend to be better suited [Beckers and Hirche, 2020].

Because dynamical systems appear in many scientific disciplines, different data-driven modeling approaches exist. These often come with their own terminology — depending on their origins — but often translate to each other. The headers in Table 2.2 give an overview of these approaches, which are also briefly described in Sections 2.2.3.1 to 2.2.3.3 below. The last column of the table, representing the operator perspective, is a central part of this thesis and is introduced in more detail in Section 2.3.

Table 2.2: Overview of modeling approaches from different disciplines.

Statistics	**Probabilistic**	**Machine learning**	**Operator theory**
• Autoregressive Moving Average Regression	• Gaussian process	• Support Vector Regression • (Deep) Neural Networks	• Extended Dynamic Mode Decomposition

Table 2.2 also includes representative methods that are used for time series modeling. The methods aim to provide a *global* description of the system. All approaches have some common ground but also key differences, meaning none of the approaches is universally suitable for system identification. Within each perspective and representative model, many extensions often exist. The edges between perspectives are soft, in that it is also common to combine benefits of different categories in new model types or generate ensembles of methods [Chan and Pauwels, 2018].

2.2.3.1 Statistics

Models coming from statistics have a strong mathematical foundation in data-driven analysis. Typically these models are not formulated in a dynamical system theory, but instead directly in data-oriented terms. A central model for time series data is the Auto-Regressive Moving Average (ARMA). The basic model assumption is that the future (univariate) system state is determined by past observed states x and errors e

$$x_{j+1} = c + \sum_{i=0}^{p-1} a_i x_{j-i} + \sum_{k=0}^{q-1} b_j e_{j-k}. \tag{2.12}$$

From the interaction of terms follows that the model has a linear structure and is, therefore, amendable for closed-form model analysis. The method explicitly accounts for temporal context in the time series by integrating time-delayed states. Moreover, probably as a result of its statistical heritage, the model acknowledges that noise is typically not i.i.d. within the temporal context. This is often disregarded by other modeling approaches.

The model has two parameters that correspond to the number of delayed states q and error terms p. Both parameters describe a family of parametric models, which is usually denoted as ARMA(p, q).

There are many extensions, which often address different temporal characteristics found in time series. This includes non-stationary processes (ARIMA), seasonality (SARMA) or vector states (VARMA). Despite their linear structure, ARMA-based models are also used in competitive time series forecasting scenarios [Makridakis et al., 2020].

2.2.3.2 Bayesian

A Bayesian (or probabilistic) perspective provides a framework for data-driven modeling that systematically handles uncertainty in the data and system [Roberts et al., 2013]. The uncertainty is also available in the model for forecasting. This is a distinct feature from other perspectives, which often only provide point estimations without indicating the model's fidelity.

A popular representative method in the Bayesian modeling is Gaussian Processes (GPs), as a powerful class for nonlinear regression tasks. GPs are often considered for static data settings [Rasmussen and Williams, 2006]. However, because of their strong foundation in probability theory and desirable features in risk-averse applications, model adaptations for system identification also exist [Beckers and Hirche, 2020; Roberts et al., 2013; Umlauft et al., 2018]. The basic model representation of a Gaussian Process for regression is

$$F_n(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), \mathcal{K}(\mathbf{x}, \mathbf{x})), \tag{2.13}$$

where the model is fully characterized by its mean function $m(\mathbf{x})$ and kernel function, $\mathcal{K}(\mathbf{x}, \mathbf{x}) : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^+$. The kernel is a central element and corresponds to a covariance function, with which the GP describes a distribution over functions. The covariance function encodes prior beliefs and can be constructed by combining simpler functions that separately describe periodicity and degree of noise [Rasmussen and Williams, 2006; Roberts et al., 2013]. By using the model in Eq. 2.13 it then becomes possible to *sample* functions, which are distributed accordingly [Umlauft et al., 2018]. Because the model is described by its available training data, the model is non-parametric as the number of parameters varies with the number of available data samples.

Typically, a single GP describes a regression function for a *single* target variable. In the formalism of a dynamical system the future state is then normal distributed Umlauft et al. [2018, Eq. 11],

$$x_{j+1} \sim \mathcal{N}(m(x_j, \sigma^2(x_j))). \tag{2.14}$$

One way to generalize this to multi-dimensional states it to concatenate N independent Gaussian Processes [Beckers and Hirche, 2020; Umlauft et al., 2018]. The drawback is that this prohibits the analysis of a single model that integrates all dynamics in a single representation. Furthermore, the nature of a dynamical system poses challenges

that complicate the accurate treatment of uncertainty [Beckers and Hirche, 2020]. This is because the recurrence of the model's output to the input creates correlations between the states, that is, a state $\mathbf{x}_{j+1}$ not only depends on $\mathbf{x}_j$ but on *all* previous states. To be computationally tractable, the number of past states considered is usually restricted, which in turn limits the simulation horizon of the model [Beckers and Hirche, 2020].

2.2.3.3 Machine learning

The term "machine learning" is often used in a broad sense to express a general data-driven modeling workflow that uses large amounts of data. This also includes the other approaches of Table 2.2. Within this section, I use the term to express a dedicated modeling perspective that combines a statistical heritage with algorithmic innovations from computer science. The "isolated" machine learning perspective in Table 2.2 provides a collection of powerful methods for regression. However, a major focus has been on *static* problem settings [Längkvist et al., 2014].

An established method that is well-founded on computational learning theory is the Support Vector Regression (SVR) [Bishop, 2006; Drucker et al., 1997]. SVR performs an often applied idea in machine learning: It transfers data measurements to a new *feature space*, where the assumption is that the regression task is well-described in a linear form. The approach led to significant contributions to learning theory that were developed by Vladimir Vapnik [see Drucker et al., 1997; Müller et al., 1997]. Like GPs, a shortcoming of the standard method description is that it only supports univariate output (however, multi-output adaptations exist [Borchani et al., 2015]).

Nowadays, conventional machine learning methods are overshadowed by a renaissance of neural network (NN) models. The evolution to so-called deep neural network (DNN) has attracted a large interest in these methods. This is mainly because the approaches often require less modeling expertise to extract meaningful features and often scale much better in terms of quality of fit with increasingly larger datasets (see LeCun et al. [2015] for a review on this matter). Oftentimes, DNNs are considered as an evolved field of machine learning, but here I consider them within the same class.

A standard feedforward NN consists of a multitude of layers. The more layers the deeper the network, where each layer can have an arbitrary number of neurons to be specified. The first and last layers correspond to the model's input and output; see Fig. 2.3. A NN ultimately corresponds to a graph, of which the weights describe a composition of simple functions to approximate a global function. During the training phase, the weights are adapted with the backpropagation algorithm.

Setting up an DNN leaves a lot of freedom to design the network structure. As a result, many diverse architectures exist, which have been designed to solve different tasks [Brunton and Kutz, 2019; LeCun et al., 2015]. While technically a DNN is parametric because it has a fixed number of parameters (the neurons), DNN have often so many neurons (in the order of millions) that they essentially appear as non-parametric.

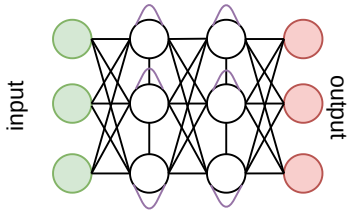


Figure 2.3: Schematic illustration of a Recurrent Neural Network (RNN) architecture. The green neurons correspond to input, white to neurons on hidden layers and red neurons to the model's output. Connections of a standard feedforward neural network are in black, whereas self-interactions specific to RNN are in purple. The figure is adapted from Brunton and Kutz [2019, Fig. 6.18]

A suitable NN architecture for addressing sequential data as in time series data are Recurrent Neural Networks (RNNs); Fig. 2.3. Unlike in a standard NN, in a RNN the neurons can self-interact. This gives rise to feedback loops and time delays as units can depend on its value at the previous time step [Brunton and Kutz, 2019]. A popular variation of RNN that solves some numerical issues and often applied for time series analysis, is the Long Short-Term Memory (LTSM) [Hochreiter and Schmidhuber, 1997].

Compared to previously mentioned methods, DNN come with a set of distinct features, making them one of the most generic approaches to approximate a function from data. DNNs are capable to capture complex nonlinear relations between high-dimensional input and output states and can scale better for highly complex tasks if more data becomes available, such as artificial intelligence (e.g. image or language recognition). The training of a DNN can be performed in a per-sample (or mini-batch) fashion, which makes it much easier to process massive datasets as individual chunks of data in an iteration. In contrast, kernel methods such as the standard SVR and GPs require all data at once in computer memory.

On the other hand, frequently mentioned drawbacks of DNN include a lack of established theory. Often the DNN architectures are based on heuristics and seem to "just work" for accurate predictions, but are otherwise black-box models that do not provide much insight to the identified system [Castelvecchi, 2016]. Moreover, DNNs usually require large datasets on which multiple epochs are processed during training. The computational cost for training is prohibitive if dedicated hardware, such as processing units on the graphic card, are not available [Thompson et al., 2020]. Furthermore, the model is typically not deterministic if the initial weights are set randomly.

2.3 Koopman operator perspective on dynamical systems

This section focuses on an operator perspective on dynamical systems as a complementary approach to those introduced in the previous section (cf. Table 2.2). At the center of the theory is the so-called Koopman operator, for which I give a brief theoretical introduction in Section 2.3.1. Because of its strong foundation in dynamical systems,

functional analysis and geometry, the methodology has become increasingly popular in recent decades [Budišić et al., 2012; Mauroy et al., 2020; Mezić, 2020].

The Koopman operator is usually computationally intractable in its exact form. However, because of its linear structure it is amenable for numerical approximation schemes. These lead to non-parametric methods that are capable to capture a wide range of nonlinear dynamics. Koopman operator-based methods have been shown to be competitive in terms of accuracy with popular LTSM models and come with significantly reduced computational requirements Eivazi et al. [2021].

Section 2.3.2 introduces the Dynamic Mode Decomposition as a numerical scheme to approximate the Koopman operator in system identification tasks. Because the final model structure is linear, the approach can reduce the gap between nonlinear dynamical systems found in practice and the well-founded mathematical understanding of linear systems. Ultimately, through a spectral form of the system it is possible to gain insight into the identified system [Avila and Mezić, 2020; Lehmberg et al., 2021; Mezić, 2020].

2.3.1 Koopman operator

In the theory of dynamical systems there is an established alternative perspective of a system's flow representation (and similarly also vector field). For every canonical dynamical system with *nonlinear* flow $F_{\Delta t}$, there is a linear operator describing the exact same system dynamics — the Koopman operator [Budišić et al., 2012; Mauroy et al., 2020; Mezić, 2020; Otto and Rowley, 2021].

The Koopman operator was first described for systems of classical mechanics by Bernhard O. Koopman [1931]. The established theory was highly relevant at the time, as it was already utilized by John von Neumann [1932] and George Birkhoff [1931] to prove the highly important "mean ergodic theorem" as part of dynamical system theory.

However, interest tapered off as evidence by the original paper being only cited only around 100 times between 1931 and 1990 [Mezić, 2020]. The discovery for a numerical treatment and its potential for data-driven modeling have renewed the interest in recent decades, which has led to a revival of the theory and an active research field [Budišić et al., 2012] (the original paper has now around 1400 citations since 1990). The seminal works on the numeric side are attributed to Igor Mezić [2005], who described the Koopman Mode Decomposition. An often cited paper of Budišić et al. [2012], providing an introduction to the theory, refers to the new research interest as "Applied Koopmanism". The overall interest is further fueled by general advances in data-driven modeling and the need to describe high-dimensional and complex dynamical systems [Reichstein et al., 2019].

In general, a linear operator describes a map between two vector spaces. The Koopman operator linearly forwards a function as element of a function space forward in time. This means that instead of describing a state evolution of a finite state $\mathbf{x} \in \mathbb{R}^N$, the Koopman operator advances functions $g(\mathbf{x}) \in \mathcal{F}$ forward in time. The change in

perspective can be understood as a coordinate transformation of the original measurement states $\mathbf{x}$ into a new function representation. During this *lift* the nonlinear state dynamics become linear in the function space representation. However, this system transformation comes also with a trade-off: To capture the full nonlinear system dynamics the Koopman operator is usually *infinite*-dimensional (i.e. $\mathcal{F}$ has an infinite-dimensional basis), even when the state space dimension is *finite* [Budišić et al., 2012]. Already simple forms of systems require infinitely many terms (Kutz et al. [2016a, Sec. 3.4] highlights this at $^{d}/_{dt}\ x = -\mu x$) and only few exceptions with a finite representation exist [e.g. Budišić et al., 2012, Example 1]. It follows that for most systems, the exact Koopman operator representation is computationally intractable. Nevertheless, the main advantage of its linear structure remains, making the operator amenable for numerical approximation schemes and data-driven modeling. The aim is to approximate and describe the operator in a computationally tractable finite function basis. A generic framework for the approximation is the Extended Dynamic Mode Decomposition (EDMD), which I describe in the next Section 2.3.2.

This section continues to describe the main theoretical statements, which often relate the equations of a finite linear dynamical system in Section 2.1. In its mathematical representation the Koopman operator, $\mathcal{U}_{\Delta t} : \mathcal{F} \to \mathcal{F}$, expresses an (autonomous) dynamical system with flow $F_{\Delta t}$,

$$[\mathcal{U}_{\Delta t} g](\mathbf{x}) = g(F_{\Delta t}(\mathbf{x})), \tag{2.15}$$

corresponding to the alternative dynamical system perspective to the integral form of the flow (Eq. 2.2). The operator acts on functions $g : \mathcal{M} \to \mathbb{C}$, which are defined on the system's state space $\mathcal{M}$. The *scalar* and complex-valued functions are referred to as *observables*. To anticipate a geometrical setting, covered in the next Section 2.4, the state space geometry $\mathcal{M}$ is assumed to be a smooth Riemannian manifold (but also generalizes to other forms, such as fractals [Otto and Rowley, 2021]).

The system representation in Eq. 2.15 is *global* and holds for *all* observables of a function space, $g \in \mathcal{F}$, at *any* state $\mathbf{x}$ [Budišić et al., 2012]. Intuitively, the observables correspond to a coordinate change in which the linear dynamics forward the system by a time step of Δt, which is equivalent to the nonlinear state evolution of the flow $F_{\Delta t}$ [Budišić et al., 2012; Kutz et al., 2016a].

A particularly relevant set of observables are those that describe the original full system state, $\mathbf{g}_{\mathrm{ID}}(\mathbf{x}) = \mathbf{x}$, which correspond to an identity function [Williams et al., 2015]. In this case it is convenient to describe the observables in a *vectorized* form of Eq. 2.15 ($\mathbf{g}$ in bolt). This means, that the Koopman operator notation of $\mathcal{U}_{\Delta t}$ is overloaded to evolve multiple (stacked) scalar observables in $\mathbf{g}(\mathbf{x})$ simultaneously (see also Budišić et al. [2012, Eq. 4]). Ultimately, this leads to the original state evolution in terms of the Koopman operator, $[\mathcal{U}_{\Delta t} \mathbf{g}_{\mathrm{ID}}](\mathbf{x}_j) = \mathbf{x}_{j+1}$.

For an easier notation, the Koopman operator representation in Eq. 2.15 is stated in terms of deterministic system dynamics. While this is rarely the case for real-world applications, the representation can be extended to a stochastic Koopman operator, in which the right hand side corresponds to the expected function values, $\mathbb{E}[g(F(\mathbf{x})]$ [Klus

et al., 2020; Mauroy et al., 2020, Eq. 13.14]. Furthermore, in favor of a follow-up numerical treatment, the observables are considered to be an element of the Lebesgue space of square-integrable functions [Otto and Rowley, 2021; Williams et al., 2015],

$$\mathcal{F} = L^2(\mathcal{M}; \mathbb{R}) := \left\{ g : \mathcal{M} \to \mathbb{R}, \int_{\mathbb{R}} |g(\tau)|^2 d\tau < \infty \right\}, \tag{2.16}$$

because it facilitates methods from function analysis, such as Galerkin approximations.

When $\mathcal{F}$ is a vector space, as in Eq. 2.16, then the Koopman operator $\mathcal{U}_{\Delta t}$ is linear [Budišić et al., 2012]. The linear structure is a key feature, because it relates to properties of finite linear systems covered in Section 2.1. For example, if the underlying system is continuous in time, then the Koopman operator has a generator,

$$\left[\mathcal{L}g\right](\mathbf{x}) = \frac{d}{dt} g(\mathbf{x}), \tag{2.17}$$

corresponding an operator representation of the vector field f in Eq. 2.1 [Klus et al., 2020, Sec. 2.1]. This also means, that a canonical time-continuous dynamical system is equipped with a semi-group of Koopman operators, describing the flow of the system at different time intervals $\Delta t > 0$ and $\mathcal{L} = \lim_{\Delta t \to 0} {}^{(\mathcal{U}_{\Delta t} - I)}/_{\Delta t}$ [Dietrich et al., 2020; Klus et al., 2020]. Note that this again relates to the statements of a standard linear system in Eq. 2.5 – 2.6.

Furthermore, the linear property allows the Koopman operator to be described in spectral terms:

$$\left[\mathcal{U}_{\Delta t} \xi_p\right](\mathbf{x}) = \xi_p(F_{\Delta t}(\mathbf{x})) = \lambda_p \xi_p(\mathbf{x}), \tag{2.18}$$

where λ_p is an eigenvalue and $\xi_p(.)$ an eigenfunction. The spectral representation has a great potential to analyze nonlinear dynamics in terms of a linear and system-intrinsic basis. For example, Mauroy and Mezić [2016] use the eigenpairs for global stability analysis. The scalar eigenvalues are now the only quantity that evolve with time; λ^j corresponding to the j-th time step. A single step of the system and its relation to the original flow is schematically depicted in Fig. 2.4.

Like for the operator itself, the dimension of the eigenbasis is generally infinite (Fig. 2.4 contains four illustrative functions). However, the "type of infinity" differs with the underlying system properties. For ergodic systems there are *countable* infinite eigenpairs, i.e. discrete point spectrum, which permit the decomposition in Eq. 2.18 [Giannakis, 2019]. For transient and chaotic systems the entire complex plane can contain eigenvalues in a continuous spectrum [Bollt, 2021; Korda and Mezić, 2020].

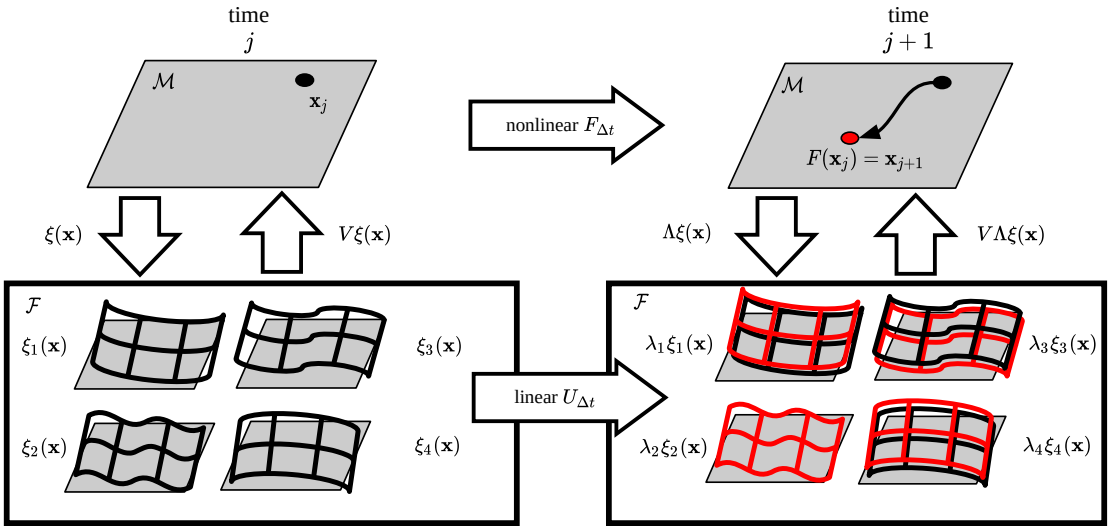


Figure 2.4: Schematic illustration of the Koopman Mode Decomposition in comparison to its nonlinear flow. Each of the Koopman eigenfunctions $\xi_p(.)$, as special observables, are defined on the state space $\mathcal{M}$. Forwarding the Koopman system in time is a simple multiplication with each corresponding eigenvalues λ_p. The Koopman modes V reconstruct the full-state observables $\mathbf{x}$ (Eq. 2.20).

Continuing with the more common assumption of a discrete point spectrum, the dynamical system can be described in terms of its eigenpairs $\{\lambda_p, \xi_p(x)\}_{p=1}^{\infty}$. The eigenfunctions, $\xi_p(\mathbf{x})$ correspond to system-intrinsic observables (i.e. scalar functions), that form a basis for the function space $\mathcal{F}$. The eigenpairs connect to the underlying state space and have interesting properties [Budišić et al., 2012; Mezić, 2020] (a list is given in Giannakis et al. [2015]).

Using the eigenbasis of the Koopman operator, any observables $g \in \mathcal{F}$ that lies in the span of the spectral basis can be expressed as

$$g(\mathbf{x}) = \sum_{p=1}^{\infty} v_p \xi_p(\mathbf{x}), \tag{2.19}$$

where the coefficients v_p are the so-called *Koopman modes* [Rowley et al., 2009, Eq. 2.4]. For the special case when $\mathbf{g}_{\mathrm{ID}}(\mathbf{x}) = \mathbf{x}$ is the (vectorized) full-state observable,

$$\mathbf{g}_{\mathrm{ID}}(F_{\Delta t}(\mathbf{x})) = F_{\Delta t}(\mathbf{x}) = \mathcal{U}_{\Delta t} \sum_{p=1}^{\infty} \mathbf{v}_p \xi_p(\mathbf{x}) = \sum_{p=1}^{\infty} \mathbf{v}_p \lambda_p \xi_p(\mathbf{x}), \tag{2.20}$$

then this describes the Koopman Mode Decomposition of the system [Mezić, 2005; Rowley et al., 2009, Eq. 2.5]; see Fig. 2.4 for an illustration. The three mathematical objects $\{(\mathbf{v}_p, \lambda_p, \xi_p(\mathbf{x}))\}_{p=1}^{\infty}$ are denoted as the *Koopman triplet* [Williams et al., 2015]. Importantly, Eq. 2.20 has the form of a linear system (cf. Eq. 2.8 on page 14) but is nevertheless able to describe a nonlinear state evolution in the full-state observables $\mathbf{x}$.

The Koopman operator has the Frobenius-Perron operator (also transfer operator) as its dual [Klus et al., 2016]. Instead of evolving observables forward in time it evolves densities, such as probability density. This makes the Frobenius-Perron operator particularly interesting for uncertainty quantification tasks in dynamical systems. Even though both operators contain the same information, the different entities have led to separate approximation techniques. However, there is also research that addresses both operators [Giannakis, 2019; Klus et al., 2016].

While the operator-based approach to dynamical systems is appealing, the data-driven approximation has to deal with the fact that both the number of functions and the point evaluations thereof are infinite in theory but finite for data-driven approximation schemes. The promise of the approach for system identification is that if the approximation of the Koopman operator is successful, it is possible describe nonlinear spatio-temporal patterns in terms of a standard linear dynamical system. The following section describes a prominent algorithm-class to achieve this.

2.3.2 Dynamic Mode Decomposition

Based on the established theory of the Koopman operator, this section navigates to a numerical and data-driven treatment. The question of how to deal with the infinite-dimensional operator based on an only finite amount of data is at the center. The theory from the previous section promises to perform a *linear* system identification in an observable space (a feature space) instead of a *nonlinear* one in the original state space. In fact, the approximation of the Koopman operator falls back to solving a linear regression problem, making the methodology particularly relevant for high-dimensional states, but also robust to noise and able to deal also with small data regimes [Karimi and Georgiou, 2021]. This greatly relaxes some of the challenges of the other approaches for system identification highlighted in Section 2.2.3. Instead of dealing with nonlinear dynamics, the main modeling challenge is now shifted to finding a suitable finite function basis in which (1) the state dynamics linearize and (2) is suitable to reconstruct the original measurement quantities.

The early works that made ground for the Koopman operator methodology in its modern data-driven setting go back to Igor Mezić [2005], who used the Koopman operator for model reduction. Schmid [2010] introduced the Dynamic Mode Decomposition (DMD) to decompose high dimensional states time series data from fluid dynamics. Rowley et al. [2009] connected DMD to the Koopman operator theory. The basic DMD can be classified as a "typical" matrix decomposition found in linear algebra

$$U_{\Delta t} \overbrace{[\mathbf{x}_1, \ldots, \mathbf{x}_{J-1}]}^{X_-} = \overbrace{[\mathbf{x}_2, \ldots, \mathbf{x}_J]}^{X_+} \tag{2.21}$$

$$U_{\Delta t}\Phi = \Phi\Lambda, \tag{2.22}$$

where X_- and X_+ are matrices that contain the data and are shifted in the index. In this standard form, the state $\mathbf{x} \in \mathbb{R}^N$ are typically assumed to be high-dimensional

with short time series (i.e. $J < N$). Notably, already at this stage $U_{\Delta t}$ is a matrix that approximates the Koopman operator (in the basis of $\mathbf{g}_{\text{ID}}(\mathbf{x} = \mathbf{x}$. This becomes clearer within the more generic numeric *extended* DMD framework detailed below.

Oftentimes, within Eq. 2.21, the states $\mathbf{x}$ are linearly reduced to a low-rank form by integrating SVD Kutz et al. [2016a, Eq. 1.18 ff]. This reduces the size of the system matrix $U_{\Delta t}$ to a manageable size if the states are high-dimensional and promotes noise terms to be truncated in the data. Eq. 2.22 performs the actual decomposition by computing the eigenbasis of the system matrix. This leads to a model

$$\mathbf{x}_{j+1} \approx \Phi \Lambda^j \mathbf{b}_1, \tag{2.23}$$

where $\mathbf{b}_1 = \Phi^{-1}\mathbf{x}_1$ are spectrally aligned initial states (alternatively, $\mathbf{b}_1 = \Phi^{\dagger}\mathbf{x}_1$ with $\dagger$ being the Moore-Penrose inverse). Overall, the DMD performs a linear spatio-temporal decomposition of time series data, which are assumed to have linear dynamics in high-dimensional states. There are various approaches for the numerical treatment, which also depends on the time series format. For an in-depth analysis in terms of a numerical linear algebra perspective I refer to Mauroy et al. [2020, Ch. 7].

The DMD relates to the Principal Component Analysis (spatial) and Fourier transformation (temporal) Kutz et al. [2016a, p. 119 ff] as other prominent data decomposition methods. But also to established methods in from data-driven modeling, such as the Hidden Markov Model (HMM) or Eigensystem Realization Algorithm (ERA) [Kutz et al., 2016a, Chapter 7]. While various approaches for the approximation exist, the DMD has become probably the most popular algorithmic class. Both the broader context in a theoretical setting and an easy-to-extend linear representation, led to numerous extensions of the DMD. The main variations in the numerical methods are to extend the problem set, address new dynamical patterns and improve the operator regression [Otto and Rowley, 2021]. Table 2.3 provides a list of selected DMD variants that seek to address different applications or aspects in dynamics. Moreover, the methodology is not isolated from ideas of the other perspectives in data driven modeling. For example, Bayesian formulations and the methods have been used to improve the convergence of neural networks [Manojlović et al., 2020], while DNN can be integrated to approximate the Koopman operator [Li et al., 2017].

Many application settings have an underlying continuous dynamical system. Instead of approximating the Koopman *operator* — relating to the flow of a system — there are also algorithms that target the Koopman *generator* [Brunton et al., 2016; Giannakis, 2019; Klus et al., 2020]. A popular algorithm is Sparse Identification of Nonlinear Dynamical systems (SINDy), which employs sparse regression to obtain a parsimonious model of a system to deduce physically informed equations [Brunton et al., 2016]. However, as highlighted in Klus et al. [2020], SINDy becomes a special case in an slightly adapted form of the extended DMD (termed gEDMD). A major drawback of approximating the Koopman generator, however, is that it requires access to time derivatives in data. These are often not available and finite difference schemes are sensitive to noise [Klus et al., 2020]. On the other hand, with the relations established in Section 2.1 for linear

dynamical system, it is also possible to obtain the Koopman generator through the operator; for details see Dietrich et al. [2020].

Table 2.3: A selection of DMD (above line) and Extended Dynamic Mode Decomposition (EDMD) (below line) variants.

Method	Short description	Reference
HankelDMD	include time-delayed states	Arbabi and Mezić [2017]
fbDMD	forward and backward dynamics	Dawson [2016]
DMD-RRR	robustly approximate random dynamical systems	Črnjarić-Žic et al. [2020]
Online DMD	update when new data becomes available	Zhang et al. [2019]
DMDc	include system control	Proctor et al. [2016]
Optimized DMD	generalize to unevenly spaced samples	Askham and Kutz [2018]
Bayesian DMD	probabilistic formulation	Takeishi et al. [2017]
MR-DMD	model dynamics at multiple scale resolutions	Kutz et al. [2016b]
EDMD-DL	learn dictionary with deep learning model	Li et al. [2017]
kernelEDMD	make use of a kernelized dictionary	Williams et al. [2014]
gEDMD	approximate the Koopman generator	Klus et al. [2020]
SINDy	discover governing equations with sparse regression	Brunton et al. [2016]
KEEDMD	approximate Koopman eigenfunctions in dictionary	Folkestad et al. [2020]

Extended Dynamic Mode Decomposition

One important and generic extension of DMD is the EDMD framework, in which the "classical DMD" is a special case [Williams et al., 2015]. The numerical steps can be explicitly described in the Koopman operator notation of the previous section.

EDMD is a Galerkin approximation of the operator, such that a finite matrix approximates the operator, $U_{\Delta t} \approx \mathcal{U}_{\Delta t}$, based on time series data. Once the matrix is available, the Koopman triplet — eigenpairs and modes — can be computed with standard linear algebra.

To cover a broader variability of sampling schemes, instead of a single time series, EDMD builds up on a collection of time series

$$X = \left[\mathbf{x}_1^{(1)}, \ldots, \mathbf{x}_{J_1}^{(1)} | \ldots | \mathbf{x}_1^{(I)}, \ldots, \mathbf{x}_{J_I}^{(I)}\right] = \left[X^{(1)}, \ldots, X^{(I)}\right] \in \mathbb{R}^{N \times \sum_i J_i}, \tag{2.24}$$

in which the column vector $\mathbf{x}_j^{(i)} \in \mathbb{R}^N$ corresponds to spatial snapshots as the system's state. The tuple of the snapshot's index (i, j) maps to a time stamp t, such that each time series is temporally ordered, $t_j^{(i)} < t_{j+1}^{(i)}$ and has a constant sampling rate, $t_j^{(i)} = t_1^{(i)} + (j-1)\Delta t$. Each state is an element of an associate multivariate time series $X^{(i)} = [\mathbf{x}_1^{(i)}, \ldots, \mathbf{x}_{J_i}^{(i)}]$, which I stack horizontally into the final matrix. Depending on the measurement modality X can be both "short and fat" (having long time series with few spatial measurements) or "tall and skinny" (having short time series with rich spatial information).

The generalization from a single coherent time series to a collection of time series goes back to Tu et al. [2014] and is also integrated in EDMD described in Williams et al. [2015]. In an extreme case the data could consist only of snapshot pairs, represented as a collection of time series each with length two.

The key idea of the approximation scheme in EDMD is to set up a finite set of observables, in which the Koopman operator is approximated. This selection is referred to as the *dictionary* of EDMD and is described as a vector-valued observable:

$$\mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}), \ldots, g_P(\mathbf{x})]^T = \mathbf{z} \in \mathbb{R}^P. \tag{2.25}$$

Together with the data in Eq. 2.24, the dictionary is integral to EDMD. The actual choice of dictionary can be seen as a "dynamic prior" and is critical for the goodness of fit because it strongly influences the quality of the model, for an example see Kutz et al. [2016a, Sec. 10.2]. The dictionary performs the coordinate change from the physical space to a feature space. While in theory the dynamics linearize with infinitely many observables $P = \infty$, the task is now to find a set of observables in which the dynamics become approximately linear with $P < \infty$. In other words, in a best case, the dictionary spans a finite function subspace of *principal* directions of the underlying infinite dimensional Hilbert space, $\mathcal{F}$ in Eq. 2.16. The approximated Koopman matrix is then an orthogonal projection onto the subspace of observables spanned by the dictionary $\mathbf{g}(\mathbf{x})$ in the limit of infinite data [Kamb et al., 2020; Williams et al., 2015]. For details on the convergence of EDMD see Korda and Mezić [2018].

Because the dictionary choice is left open, EDMD is a highly flexible and generic framework to realize a *computable* Koopman theory. Some DMD variations are special cases of EDMD. There are different strategies of how to set up the dictionary, such as making use of prior knowledge or using generic methods that extract a suitable function basis from data [Giannakis, 2019]. A good dictionary includes "informative" observables which are usually not known *a priori* and depend on several factors, such as the underlying system or the sampling strategy [Williams et al., 2015]. The actual dictionary choice is therefore frequently stated to be an open problem, since the task is comparable with the general "model selection" procedures in machine learning [Kutz et al., 2016a; Surana, 2020]. But also computational costs — memory or computation time — can be factors. The problem of dictionary choice has also attracted follow-up research that try to "learn the dictionary" to automatically approximate a wider range of systems; for example EDMD-DL in Table 2.3.

The following block of mathematical statements contains the main numerical steps of EDMD to set up a linear system and describe it in its spectral terms. Each statement is then described in more detail. For an easier notation, I use the same corresponding symbols as for the exact Koopman theory, but now mean to express finite representations and data-inferred approximations of the mathematical objects.

1. Given: time series collection X (Eq. 2.24) and dictionary $\mathbf{g}(\mathbf{x})$ (Eq. 2.25).

2. Apply dictionary to available data to obtain feature state matrix Z and compute linear map $B\mathbf{z} = \mathbf{x}$

$$\mathbf{g}(X) = \left[\mathbf{z}_1^{(1)}, \dots, \mathbf{z}_{J_1}^{(1)} | \dots | \mathbf{z}_1^{(I)}, \dots, \mathbf{z}_{J_I}^{(I)}\right] = Z \tag{2.26}$$

$$B = XZ^{\dagger} \tag{2.27}$$

3. Approximate Koopman operator $\mathcal{U}_{\Delta t} \approx U_{\Delta t} \in \mathbb{R}^{[P \times P]}$ in a least-squares sense with time shift matrices

$$U_{\Delta t} = Z_+ Z_-^{\dagger} \tag{2.28}$$

$$Z_+ = \left[\mathbf{z}_2^{(1)}, \dots, \mathbf{z}_{J_1}^{(1)} | \dots | \mathbf{z}_2^{(I)}, \dots, \mathbf{z}_{J_I}^{(I)}\right]$$
$$Z_- = \left[\mathbf{z}_1^{(1)}, \dots, \mathbf{z}_{J_1-1}^{(1)} | \dots | \mathbf{z}_1^{(I)}, \dots, \mathbf{z}_{J_I-1}^{(I)}\right]$$

4. Diagonalize Koopman matrix with eigenvalue matrix $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_P)$ and the right and left eigenvectors respectively (Φ, Φ^{-1})

$$U_{\Delta t} = \Phi \Lambda \Phi^{-1} \tag{2.29}$$

For a clearer notation, in Eq. 2.26, I overload the dictionary function of Eq. 2.25 to map the data matrix X to a feature matrix Z, i.e. the dictionary is applied for each snapshot. Because we are ultimately interested in the measurement state evolution of $\mathbf{x}$, we also compute a matrix B, which linearly maps feature states back to the physical states (the symbol † denotes the Moore-Penrose inverse). Notably, this means that the dictionary state representation $\mathbf{z}$ should be suitable for both linearly describing the dynamics and reconstructing the measurements.

Eq. 2.28 sets up two time shifted matrices in the feature space, such that in the matrix Z_+ every first state of a series and Z_- every last state is dropped accordingly. The Koopman matrix is then computed in a least squares sense between these two matrices, mapping from $\mathbf{z}_j$ to $\mathbf{z}_{j+1}$.

At this stage the system can be already stated in a linear state space representation, corresponding to the general system identification form in Eq. 2.10 – 2.11. The intrinsic form $F(\mathbf{z}_j) = U_{\Delta t}\mathbf{z}_j = \mathbf{z}_{j+1}$ — relating to the operator representation in Eq. 2.15 — describes the dynamics and the original state evolution is reconstructed with the output matrix, $G(\mathbf{z}_j) = B\mathbf{z}_j = \mathbf{x}_j$.

However, we continue to decompose the system into its favorable spectral representation. In Eq. 2.29, the Koopman matrix is diagonalized, allowing us to set up the final model

$$\mathbf{x}_{j+1} \approx B\left(U_{\Delta t}^{j}\mathbf{z}_1\right) \tag{2.30}$$

$$\approx B\left(\Phi\Lambda_{\Delta t}^{j}\Phi^{-1}\mathbf{z}_1\right) \tag{2.31}$$

$$\approx V\Lambda_{\Delta t}^{j}\xi\left(\mathbf{x}_1\right) = \sum_{p=1}^{P}\mathbf{v}_p\lambda_p^j\xi_p\left(\mathbf{x}_1\right), \tag{2.32}$$

where the last equation corresponds to the Koopman Mode Decomposition [Mezić, 2005; Rowley et al., 2009]. The matrix $V = B\Phi$ contains the Koopman modes and reconstructs the states, whereas $(\lambda_p, \xi_p(\mathbf{x}))_{p=1}^{P}$ correspond to the approximate Koopman eigenvalues and eigenfunctions. In the discrete map a single iteration, $j \to j+1$, associates to the sampling interval Δt. Again the single terms can be mapped to the general system identification form in Eq. 2.10 – 2.11.

Because Eq. 2.32 describes an autonomous and finite linear dynamical system there is a unique and analytical solution for every initial condition. Moreover, the stability criteria of Table 2.1 on page 15 apply. This means that the prediction is now fully determined by the Koopman triplet and oscillates according to the imaginary part in eigenvalue and either grows, decays or stays stable over time depending on the magnitude of the eigenvalue. Moreover, it is possible to select only principal triplets, which leads to reduced order models [Mauroy et al., 2020].

To perform a prediction with Eq. 2.32 it is required to evaluate the P Koopman eigenfunctions at the initial conditions, $\xi : \mathbb{R}^N \to \mathbb{C}^P$ (ξ_p corresponds to the p-th coordinate in Eq. 2.32). Importantly, this mapping should also be available for states $\mathbf{x}$ that are not contained in the data used to set up the model. The (vectorized) eigenfunctions require the evaluation of the dictionary and spectrally align it with the left eigenvectors of the Koopman matrix, $\xi(\mathbf{x}) = \Phi^{-1}\mathbf{g}(\mathbf{x}), \mathbf{x} \notin X$.

In case $U_{\Delta t}$ is not diagonalizable in Eq. 2.29, it is also possible to compute generalized eigenfunctions in a Jordan block-form, for details see Korda and Mezić [2020]. Another way to circumvent computing the left eigenvectors is to approximate the Koopman eigenfunctions in a least-squares sense $\xi(\mathbf{x}) = \Phi^{\dagger}\mathbf{g}(\mathbf{x})$, which only requires the right eigenvectors [Kutz et al., 2016a].

In the current formulation the dictionary is left intentionally open. The next section describes data transformations and functions that relate to the underlying geometry of time series data. As such they can provide a well-defined and data-adaptive function basis for the Koopman operator.

2.4 Geometry in time series data

The success in data-driven modeling often depends on finding a suitable representation of data [Bengio et al., 2013]. In the context of system identification, it is essential to find a state representation that is suitable to describe the spatio-temporal patterns of the system. This is highlighted in the general system identification form in Eq. 2.10 – 2.11 (page 17), in which the system dynamics are defined on a separate (intrinsic) state variable $\mathbf{z}$ and not on the original measurements $\mathbf{x}$. Since the measurement data is the main source of knowledge, the data transformation can be described as

$$\mathbf{g}(\mathbf{x}) = \mathbf{z}, \tag{2.33}$$

in which both $\mathbf{g}(\mathbf{x})$ and target values $\mathbf{z}$ have to be inferred from data. In a machine learning terminology this corresponds to an unsupervised task.

In Eq. 2.33 I re-use the notation of the EDMD dictionary of the previous section. This highlights the equivalence to EDMD where $\mathbf{z}$ expresses a suitable new state representation which in the best case has the quality of having (approximately) linear dynamics. In fact, the methods covered in this section are later combined with the Koopman operator approach with exactly this objective.

In this section, I continue to find such meaningful intrinsic state representations from measurement data. Unlike for learning the system dynamics itself this task is much less well-defined. This is because there is no obvious way of how to describe the data or extract "interesting patterns" [Murphy, 2012]. For setting up the function $\mathbf{g}(\mathbf{x})$ there is no error metric available since the target states $\mathbf{z}$ itself are unknown. It is therefore not possible to quantify of how suitable a new representation is.

There are different approaches for representation learning. Here I highlight data transformations that are motivated by an *geometric* understanding of the time series data and the underlying system. This section includes both a *spatial* and *temporal* feature extraction method to obtain a new state representation. Ultimately, both methods have a solid theoretical foundation, are noise robust, have strong analytical power and connect well to the Koopman operator.

In Section 2.4.1, I first provide the notation of a neighborhood graph as a basis to describe nonlinear geometry in static point clouds. This setting is then used in Section 2.4.2 to extract a geometrically aligned function basis. Section 2.4.3 extends the geometric understanding to the time delay embedding as a method to reconstruct dynamics from measurement time series. Finally, Section 2.4.4 discusses the composition of the two methods to obtain a *spatio-temporal* state representation and gives links to research that combines the main methods described in this thesis.

2.4.1 Neighborhood graphs: A basis for explicit manifold learning

Many data-driven methods only contain an *implicit* description of identified geometric patterns of the underlying process. These are hidden in the model's parameters and, therefore, hard to relate to the modeled system or interpret. Other popular methods

from linear algebra have a clear description of geometry (e.g. Principal Component Analysis), but have the too restrictive assumption of a linear geometry, which is mostly not satisfied in practice. This section introduces the foundation of kernel-based algorithms, aiming to give an *explicit* description of nonlinear geometry in data [Belkin and Niyogi, 2008; Belkin et al., 2009; Berry and Sauer, 2019; Coifman and Lafon, 2006b].

The geometry in a data-generating system, can be described with the mathematical notion of a *manifold*. Here, I only provide an introduction of a manifold by outlining the most important aspects to convey an intuitive understanding. This is mostly sufficient for a data-driven modeling, in which the manifold remains unknown. Both Bishop [2006] and Murphy [2012] as common reference books for machine learning use the term "manifold", but never give a detailed explanation. For an in-depth introduction, I refer to Lee [2012]; Ma and Fu [2012].

A (compact) manifold is a topological space, that can be imagined as an n-dimensional nonlinear (i.e. curved) surface that *locally* resembles Euclidean space (homeomorphic, in mathematical terms) [Lee, 2012]. Fig. 2.5 illustrates a concrete example manifold. The blue-wired structure describes the so-called "swiss-roll" as an easy-to-analyze geometry that is often used in literature. The swiss-roll has a dimension of $n = 2$, but is itself embedded in a $N = 3$ dimensional ambient space. The local property of a manifold states that a patch — corresponding to a blue cell in the wire — is an "almost flat" two-dimensional surface (i.e. homeomorphic to $\mathbb{R}^2$). For the swiss-roll it is possible to extract two geometrically aligned coordinates, which describe the position on the manifold. The reduction from three to two coordinates corresponds to an *embedding*. In this case it can be imagined as an "unrolling" of the swiss-roll (this is performed in Section 2.4.2).

An example of an invalid manifold is when the geometry crosses with itself. This would be the case in a geometry that forms a figure "eight" (8). The problem lies at the intersection point, because the local space around this point is not flat anymore and therefore does not resemble Euclidean space. If the geometry describes a state space for a dynamical system, it would not be clear which "branch" to follow at the intersection. Furthermore, in other settings with a valid manifold it is not always possible to embed it into a space that corresponds to its intrinsic dimension. For example, if a geometry "closes" on itself, such as a line that forms a "zero" (0), can only be described in two coordinates, despite its intrinsic dimension of one. An embedding into one coordinate would require a "cut", which then no longer describes the local geometry around the cut.

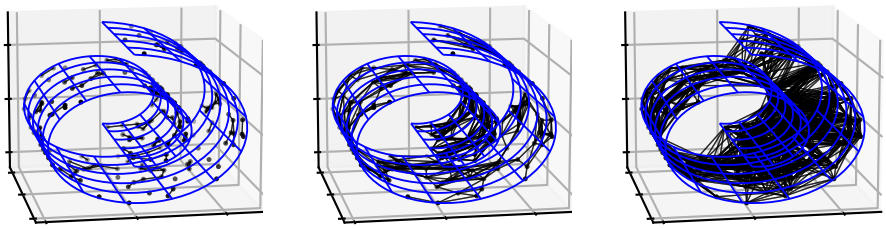


Figure 2.5: The swiss-roll manifold. The wired (blue) grid corresponds to the true manifold on which points are sampled on. Each of the plots includes a neighborhood graph to capture the local geometry of the manifold, with too few (left), well-suited (middle) and too many connections (right).

A main challenge in learning a manifold from data is that there are only a finite number of points available. Here I drop the temporal context of a time series and only consider a *static* point cloud. The aspects seamlessly transfer to time series data in that the additional time information is carried along.

The typical assumption is that the available samples in a dataset, $\mathbf{x} \in X \in \mathbb{R}^{[J \times N]}$, are sampled directly (or near, in the presence of noise) on an unknown manifold $\mathcal{M}$. The manifold itself is embedded in the ambient (measurement) space, $\mathbf{x} \in X \subset \mathcal{M} \subset \mathbb{R}^N$. This is also referred to as *manifold assumption*. In the example of Fig. 2.5, $\mathbf{x} \in \mathbb{R}^3$ is a point in the ambient space, X the point cloud, and $\mathcal{M}$ the true underlying swiss-roll geometry. The goal of *manifold learning* — a way of geometrically-informed representation learning — is to find and extract geometric coordinates, without loosing significant structural information. Two major use cases are to obtain a new basis of the data that becomes suitable for further processing [Berry et al., 2015; Giannakis, 2019; Lehmberg et al., 2021] and to select principal coordinates for nonlinear dimension reduction if $n \ll N$ [Coifman and Lafon, 2006b]. Through the explicit and sorted representation of the manifold it is possible to gain insight into and interpret the data-generating process.

A central procedure of many manifold learning methods is setting up a *neighborhood graph* on the point cloud [Strange and Zwiggelaar, 2014]. Ultimately, the graph acts as an empirical surrogate of the manifold and, as Coifman and Lafon [2006b, p. 5] state, "*offer an advantageous compromise between their simplicity, interpretability and their ability to represent complex relationships between data points*". A crucial aspect in setting up the neighborhood graph is that it acknowledges the local property of a manifold. This is typically achieved by giving close-distanced (in Euclidean metric) point pairs a larger weight than further distanced point pairs, if they are in a neighborhood relation. In contrast, point pairs that are *not* in a neighborhood relation have no connection. Ultimately, the graph should span the entire point cloud and provide a new orientation on the geometry. Instead of measuring the Euclidean distance in the ambient space, it is now possible to take the shortest path on the graph as a way to measure the *geodesic dis-*

tance along the curved geometry [cf. Isomap method in Balasubramanian, 2002]. If the graph includes "false connections" then these shortcut the true geodesic distance and lead to misleading representations. This is exemplified in the Fig. 2.5, where the right plot includes too many connections and represents the geometry poorly. On the other left side it is also possible to include too few connections. This can lead to disconnected neighborhood graphs in which there is no relation between all point pairs in the point cloud. Only the middle plot is well-connected and represents the swiss-roll manifold well.

The important follow-up question is, how to set up the graph algorithmically. For this task a *kernel* is introduced. A kernel gives a weight for each pairwise sample $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$, which can also be seen as similarity measure between the point pair. A kernel has the following properties [Coifman and Lafon, 2006b]:

- symmetric, $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) = \mathcal{K}(\mathbf{x}_j, \mathbf{x}_i)$
- positive, $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) \geq 0$

Often the actual choice of a specific kernel is left open and can be specified for the concrete application. This provides a way to include prior knowledge about the similarity between points. The kernel then corresponds to a "geometric prior" because it induces the local geometry of the inferred manifold [Berry and Sauer, 2016; Coifman and Lafon, 2006b]. If no specific domain knowledge is available, there is a set of generic priors that can be used [Bengio et al., 2013]. A typical standard kernel in manifold learning is the Gaussian kernel

$$\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{d(\mathbf{x}_j, \mathbf{x}_i)^2}{2\varepsilon}\right), \tag{2.34}$$

with a Euclidean distance function $d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_j - \mathbf{x}_i\|_2$ and bandwidth parameter ε. So far, the kernel only maps a weight to each point pair, but does not actually exclude any connection in the graph. However, kernels that exponentially decay for large distance values, such as the Gaussian in Eq. 2.34, assign negligible weights for far-distanced points. Given a suitable bandwidth ε, even if the actual neighborhood graph is fully connected, it can essentially describe a similar graph to the one in the middle of Fig. 2.5.

Based on the notion of a kernel, we can now describe the neighborhood graph in a matrix form $K \in \mathbb{R}^{[J \times J]}$. Each element in the matrix describes the pairwise weight. Given the above properties of a kernel, the matrix K is symmetric and positive semi-definite.

Algorithmically, it is suitable to separate the construction of the kernel matrix into two main steps [Strange and Zwiggelaar, 2014].

1. Compute distance matrix:

$$d_{i,j} = \begin{cases} d(\mathbf{x}_i, \mathbf{x}_j) & \text{, if } \mathbf{x}_i \text{ and } \mathbf{x}_j \text{ are in neighborhood relation} \\ \infty & \text{, else} \end{cases} \tag{2.35}$$

2. Compute neighborhood graph with kernel function:

$$K_{i,j} = \mathcal{K}(d_{i,j}) = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) \tag{2.36}$$

Note that in the second step, the kernel function is overloaded and performs the evaluation on an already computed distance value in the argument.

The advantage of this separation is a dedicated treatment of the distance matrix in the first step. This allows an early exclusion of far-distanced connections, and gives rise to sparse neighborhood graphs. As highlighted above the commonly used exponentially decaying kernels can also provide reasonably neighborhood graphs (despite being fully connected). However, adopting sparse matrices can also be motivated by limited computing resources. A full distance matrix has memory requirements that scale quadratic with the number of samples in the dataset, $\mathcal{O}(J^2)$, and similarly the number of floating point operations $\mathcal{O}(J \cdot N)$.

There are two primary approaches to compute a sparse distance matrix (leading to a sparse kernel matrix). The first is the k-nearest neighborhood (k-NN) which stores a fixed number neighboring points per sample. The second is a δ-range neighborhood graph (δ-range) which includes all neighboring points that are within a ball of radius δ, centered at the respective point Strange and Zwiggelaar [2014, Sec. 3.1]. Both approaches have different qualities, which are summarized in Table 2.4.

Table 2.4: Overview of characteristics between k-NN and δ-range neighborhood graph.

	k-NN	**δ-range**
sparsity	fixed	variable
outliers	connected	disconnected
sparse region	far-distanced neighbors	few neighbors
dense region	short-distanced neighbors	many neighbors
symmetry	non-symmetric	symmetric

The k-NN includes a fixed number of neighbors per sample, whereas the δ-range has a variable number and, therefore, a less predictable sparsity. The graph of k-NN is always connected, whereas outliers in δ-range can be disconnected. The two algorithms also handle varying sampling densities differently. In k-NN sparsely sampled regions become apparent if neighbors are far-distanced, in δ-range there are less neighboring points included. This relates to densely sampled regions accordingly. Finally, given the two properties the final kernel matrix of k-NN is non-symmetric and symmetric for δ-range.

Both methods require a careful selection of their respective parameters k and δ to describe a suitable neighborhood relation, such as exemplified in middle plot of Fig. 2.5. A challenge, however, are varying degrees of point densities on the manifold. Selecting a single *global* parameter to describe the entire point cloud then often represents a trade-off.

For both methods there is extensive literature and also many extensions. These extensions are often motivated by storing the point cloud in tree-based data structures (coming from computer science) that have already a coarse neighborhood relation which can limit the number of pairwise evaluations [Muja and Lowe, 2014].

A kernel matrix K is the foundation of spectral methods that aim to extract geometric information. Ultimately, the construction requires making choices about how to define the neighborhood via the connections (distance algorithm, $d(\mathbf{x}_i, \mathbf{x}_j)$ and weights (kernel, $K(d_{i,j})$). The optimality of the choices are problem specific and depend on the underlying pairwise distances and point distribution. Different choices can lead to fundamentally different manifold extractions [Berry and Sauer, 2016].

2.4.2 Laplace-Beltrami operator

This section continues to robustly extract geometrical coordinates from a neighborhood graph (kernel matrix) set up in the previous section. In particular the class of "kernel eigenmap methods" manipulate a kernel matrix in certain ways and then compute the eigenvectors [Belkin and Niyogi, 2008; Coifman and Lafon, 2006b; Strange and Zwiggelaar, 2014]. The set of eigenvectors then describe new latent variables that adapt to the underlying geometry. This can be understood as a nonlinear projection $\mathbf{g}(\mathbf{x})$ from the original point cloud onto the eigenvectors $\mathbf{z}$. At best the new eigen-coordinates preserve the local geometrical structure of the true underlying manifold [Coifman and Lafon, 2006b]. Examples of methods in this class include the Local Linear Embedding [Roweis and Saul, 2000], Laplacian Eigenmaps [Belkin and Niyogi, 2007], Hessian Eigenmaps [Donoho and Grimes, 2003] and Diffusion Maps (DMAP) [Coifman and Lafon, 2006b].

Compared to general (unsupervised) representation learning, kernel eigenmap methods promote a sound theoretical foundation to describe geometry. In particular, the method variations in eigenmap methods have increasingly focused on estimating the Laplace-Beltrami operator [Belkin and Niyogi, 2007, 2008; Berry and Giannakis, 2020; García Trillos et al., 2020]. The operator generalizes the usual Laplace operator as the divergence of the gradient to manifolds, $\Delta_{\mathcal{M}} f = \mathrm{div}(\nabla_{\mathcal{M}} f)$. The Laplace-Beltrami operator has a strong foundation in differential geometry and encodes important geometric information of Riemannian manifolds on which it is defined [Belkin et al., 2009; Berry and Giannakis, 2020; Berry and Harlim, 2016]. Because the Laplace-Beltrami operator is linear, its primary interest lies in finding its spectral components, that is, the eigenfunctions and eigenvalues.

Fig. 2.6 illustrates an example of approximated eigenfunctions of Laplace-Beltrami operator for the swiss-roll manifold. Similar to Principal Component Analysis (PCA) the eigenvalues provide an order of importance and associate to the length of the coordinate that is aligned on the manifold [Dsilva et al., 2018]. A central objective in research of kernel eigenmaps is to process the neighborhood graph such that it becomes *consistent*, that is, that it converges to the Laplace-Beltrami operator in the limit of large data

in a variety of data settings [Belkin and Niyogi, 2007; Berry and Sauer, 2019; Coifman and Lafon, 2006b].

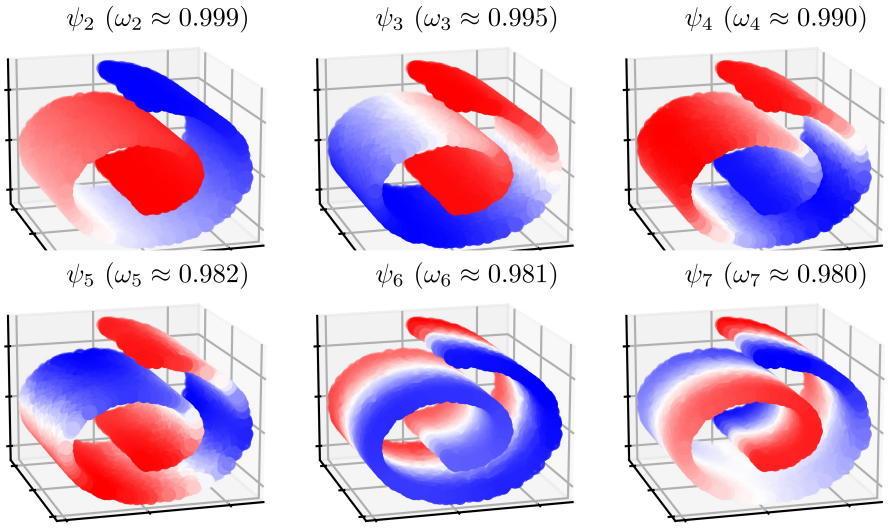


Figure 2.6: Example of the first six approximated eigenfunctions of the Laplace-Beltrami operator on the swiss-roll manifold. The dataset includes 5000 samples and is approximated with the DMAP on a neighborhood graph with Gaussian kernel of bandwidth $\varepsilon = 6$ (cf. Eq. 2.34) and a sparse δ-range distance matrix with $\delta = 3$ (in Euclidean metric). The color-code encodes the variation of the respective function from minimum (blue) to maximum (red) values. A script to generate the plot is contained in the Supplementary Material.

There are a range of interesting applications, in which the Laplace-Beltrami operator becomes useful. In this thesis, the main focus lies on the following two use cases:

1. **Nonlinear dimension reduction** The typical *manifold assumption* states that high dimensional states in an *ambient* space are in fact sampled on a much lower dimensional manifold [Lin et al., 2015]. The eigenfunctions of the Laplace-Beltrami operator are suitable to "disentangle" different manifold directions in the point cloud and provide them in a hierarchical order (given by the eigenvalues) [Bengio et al., 2013; Coifman and Lafon, 2006b]. Nonlinear dimension reduction becomes possible if the eigenfunctions are seen as a set of new coordinates in which the high-dimensional data can be embedded in [Coifman and Lafon, 2006b]. For dynamical systems, these coordinates can then provide dynamically meaningful reduced (or macroscopic) variables [Dsilva et al., 2016, 2018; Nadler et al., 2006].

   An important task in the nonlinear dimension reduction is to keep the essential geometrical structures relevant for the application. However, unlike for linear dimension reduction (e.g. PCA), finding the suitable set is not straightforward. Instead of truncating low-order coordinates solely based on their eigenvalue, nonlinear dimension reduction often requires an additional selection process. This is because

there can be "repeated eigendirections" [Dsilva et al., 2018]. In Fig. 2.7 this becomes apparent where the first four eigenfunctions, ψ_{2-5}, have varying oscillations but are otherwise aligned in the same direction (along the long side of the swiss-roll). Only ψ_6 aligns a new and independent axis along the short side of the swiss-roll.

In Fig. 2.7 I perform a data embedding on the coordinates of Fig. 2.6. Based on these embeddings we can see that $(\psi_2, \psi_6$ does in fact "unroll" the swiss-roll to two dimensions. The two coordinates align to the length and width (cf. Fig. 2.6). The other displayed combinations collapse to a one-dimensional embedding, which results in a loss of essential geometric information ($\psi_{3,4,5}$ repeat the direction of ψ_2. For an automatic way to select coordinates and a more in-depth discussion, see [Dsilva et al., 2018].

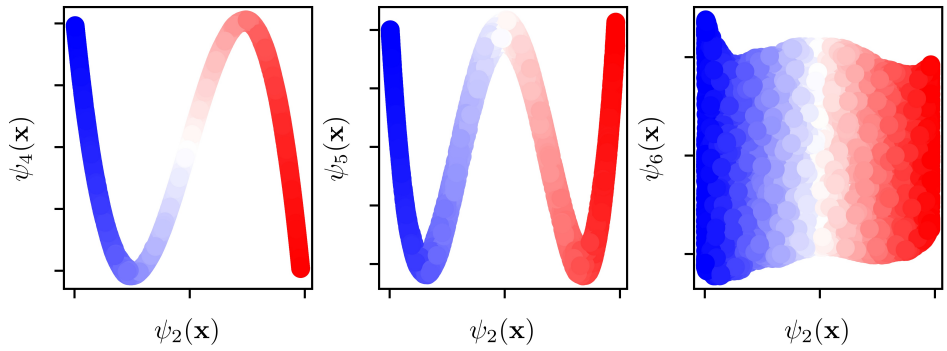


Figure 2.7: A selection of possible embeddings to project the three-dimensional points samples of the swiss-roll. The color-code corresponds to the function values of ψ_2, which describe the length of the swiss-roll (cf. upper left graph in Fig. 2.6).

2. **Specify a function basis on the manifold** In the context of this thesis, which analyses time series, I assume that the time series data is spatially and temporally coherent, forming a single and connected manifold [Bengio et al., 2013]. This contrasts, for example, to applications of the methodology, where one seeks to cluster separate point clouds into their respective class [Bengio et al., 2013; Coifman and Lafon, 2006b]. A further assumption about the underlying manifold is that it is smooth, meaning that a tangent plane on the manifold varies continuously from point to point. These topological structures enable defining general functions on the manifold [Coifman and Lafon, 2006a; Lee, 2012].

   Of particular interest are the eigenfunctions of the Laplace-Beltrami operator itself, because they form a function basis for square integrable functions on the manifold ($L_2(\mathcal{M}, \mathbb{R})$, cf. Eq. 2.16) [Belkin et al., 2009]. In fact, the eigenfunctions correspond to so-called geometric harmonic functions, that are analogous to Fourier modes[1] and refer to generalized sine and cosines on the manifold (analogous to Fourier modes)

[1] Because of the connection between harmonic functions and geometry, the question "Can one hear the shape of a drum?" was raised in https://www.math.ucdavis.edu/~hunter/m207b/kac.pdf

[Belkin et al., 2009; Berry et al., 2013]. In Fig. 2.6 the harmonic waves can be visualized for the swiss-roll manifold. This quality is particularly interesting in the context of observables in the Koopman operator [Berry and Sauer, 2016; Giannakis, 2019].

Algorithmic framework of Diffusion Maps (DMAP)
To approximate the Laplace-Beltrami operator, I focus on the DMAP framework. The method performs normalization steps on the kernel matrix to that are suitable to obtain a consistent graph for more general data settings compared to the other kernel eigenmap methods [Coifman and Lafon, 2006b]. For example, DMAP is noise robust and can de-bias the point density as an artifact of the data collection. This maintains consistency for arbitrary point distributions on Riemannian manifolds [Coifman and Lafon, 2006b]. The method has become probably the most popular kernel eigenmap method, because it is the subject of ongoing research and adaptations to improve the convergence to the Laplace-Beltrami operator [Berry and Giannakis, 2020, Table 1.1] [Coifman and Lafon, 2006b; García Trillos et al., 2020]. While the DMAP has its origins in nonlinear dimension reduction of static data, it has also been integrated into the analysis and modeling of dynamical systems in follow-up research [e.g. Alexander and Giannakis, 2020; Berry et al., 2013; Dietrich et al., 2016; Giannakis, 2019; Kemeth et al., 2018].

The following block of mathematical statements describe the DMAP. All equations of the framework refer to the discrete case of processing finite data X; the continuous counterparts for a theoretic treatment are contained in the original work of [Coifman and Lafon, 2006b].

1. Given: data $X \in \mathbb{R}^{[J \times N]}$ with manifold assumption $X \subset \mathcal{M}$, distance function $d(.,.)$ (Eq. 2.35), kernel $\mathcal{K}$ (Eq. 2.36), re-normalization parameter $\alpha \in [0, 1]$

2. Compute kernel matrix

$$K_{i,j} = \mathcal{K}(d(\mathbf{x}_i, \mathbf{x}_j)), \forall (\mathbf{x}_i, \mathbf{x}_j) \in X \tag{2.37}$$

3. Normalize sampling density (P is a diagonal matrix)

$$K^{(\alpha)}_{i,j} = P^{-\alpha} K P^{-\alpha}, \text{ with } P^{-\alpha}_{i,i} = \sum_j K_{i,j} = q(\mathbf{x}_i) \tag{2.38}$$

4. Describe a diffusion process in a symmetric matrix, $M^{(\text{con})}$, which is conjugate to the non-symmetric Markov matrix diffusion process:

$$M^{(\text{con})} = S^{-1/2} K^{(\alpha)} S^{-1/2}, \text{ with } S_{i,i} = \sum_j K^{(\alpha)}_{i,j} \tag{2.39}$$

5. Compute eigenpairs $\{\mathbf{w}_p, \omega_p\}_{p=1}^{P}$ sorted by eigenvalues in descending order and recover eigenvectors of Markov matrix:

$$M^{(\text{con})} \mathbf{w}^{(\text{con})}_p = \mathbf{w}^{(\text{con})}_p \omega_p \tag{2.40}$$

$$\mathbf{w}_p = S^{-1/2} \mathbf{w}^{(\text{con})}_p \tag{2.41}$$

6. Define out-of-sample vectors for samples $\mathbf{x}_{\text{oos}} \notin X$ ($[.]_i$ describes the i-the component of a vector):

$$[K^{(\alpha)}_{\text{oos}}]_i = \frac{\mathcal{K}(\mathbf{x}_{\text{oos}}, \mathbf{x}_i)}{q(\mathbf{x}_{\text{oos}}) q(\mathbf{x}_i)}, \forall \mathbf{x}_i \in X \qquad [M_{\text{oos}}]_i = \frac{[K^{(\alpha)}_{\text{oos}}]_i}{\sum_j [K^{(\alpha)}_{\text{oos}}]_j} \tag{2.42}$$

7. Define Nyström [1930] extension [see also Rabin and Coifman, 2012]

$$\psi_p(\mathbf{x}) = \frac{1}{\omega_p} \langle M_{\text{oos}}, \mathbf{w}_p \rangle \tag{2.43}$$

8. Define DMAP embedding with P computed coordinates, $\mathbf{g}_{\text{dmap}} : \mathbb{R}^N \to \mathbb{R}^P$

$$\mathbf{g}_{\text{dmap}}(\mathbf{x}; t) = [\omega_1^t \psi_1(\mathbf{x}); \omega_2^t \psi_2(\mathbf{x}); \ldots; \omega_P^t \psi_P(\mathbf{x})], \text{ for } \mathbf{x} \in \mathcal{M} \tag{2.44}$$

For an easier notation, I omit the time reference in the data, where the i-th sample is $\mathbf{x}_i \in X$. If X is a time series collection, the indices and temporal order is maintained during the transformation such that $\mathbf{x} := \mathbf{x}_j^{(i)}$ (using the index notation in Eq. 2.24).

Statements 1 and 2: The kernel function $\mathcal{K}(d(.,.))$ with the distance in the argument follows the notation of setting up a neighborhood graph, described in the previous section. The choice of kernel can be task-specific and is flexible in the framework [Coifman and Lafon, 2006b]. The theoretical work for DMAP was extended in Berry and Harlim [2016] to so-called *local* kernels. The Gaussian kernel in Eq. 2.34 is the usual default and is itself parametrized with a kernel bandwidth ε. However, there are also interesting alternatives such as "continuous nearest neighbor" kernel, which leads to a sparse and (unweighted) adjacency graph that still converge to the Laplace-Beltrami operator [Berry and Sauer, 2019]. It is also possible to include a temporal context in "cone-kernels", which give more weight to samples that lie in the direction of the flow [Giannakis, 2015].

Statements 3 to 5: Based on the available kernel matrix K, Eq. 2.38 corresponds to a normalization of the sampling density. The density values, $q(\mathbf{x}_i)$, are on the diagonal matrix of P, of which the normalization degree is steered by the parameter α. At one end, $\alpha = 0$, disables the correction of the sampling density ($P = I$ corresponding to the identity matrix). This corresponds to the typical graph Laplacian and requires uniform sampling on the manifold to be consistent. At $\alpha = 0.5$ the Fokker-Plank operator is approximated and at the other end, $\alpha = 1$, the normalization removes the influence of the sampling density. The latter case recovers the Riemannian geometry of compact manifolds for arbitrary sampling distributions in the data [Coifman and Lafon, 2006b].

The core idea of DMAP in the next step is to compute a suitably normalized row-stochastic Markov matrix M, to describe a diffusion process. An entry $M_{i,j}$ corresponds to a diffusion probability from sample $\mathbf{x}_i$ to $\mathbf{x}_j$. However, the standard left normalization to a row-stochastic matrix, $M = S^{-1}K^{(\alpha)}$, leads to a non-symmetric matrix M.

Instead, statement 4 performs a similarity transformation, $S^{-1/2}M^{(\text{con})}S^{1/2} = M$ such that $M^{(\text{con})}$ is symmetric and *similar* to M (i.e. shares same eigenvalues and eigenvectors can be recovered in Eq. 2.41). For a more detailed description on the conjugate matrix in DMAP see Berry et al. [2013].

Eq. 2.40 then computes the eigenpairs of the manipulated matrix $M^{(\text{con})}$. Computationally, this operation now benefits from the similarity transformation, because dedicated eigensolver algorithms for symmetric matrices are more stable and robust [Hernandez et al., 2009]. Moreover, the matrix is positive semi-definite, due to its relation to the original Markov matrix. The largest eigenvalue is $\omega_1 = 1$ with an associated constant eigenvector and all subsequent eigenvalues are real-valued and decrease towards zero $\omega_i \to 0$ for $i \to \infty$. To process larger datasets this allows computing only the first P leading eigenpairs and truncate pairs with eigenvalue near zero.

An eigenvector's i-th element corresponds to the point evaluation of the approximate eigenfunction of the Laplace-Beltrami operator, $[\mathbf{w}_p]_i = \psi_p(\mathbf{x}_i)$. For $\alpha = 1$, $N \to \infty$ and kernel bandwidth $\varepsilon \to 0$ the eigenvectors converge to the operator for compact manifolds [Coifman and Lafon, 2006b, Proposition 3].

Statements 6 and 7: From the eigenvectors in the previous step only a finite set of function evaluations are available. However, for practical relevance in a machine learning setting, it is essential to extend the function to arbitrary points in the neighborhood of the point cloud, i.e. $\psi_p(\mathbf{x})$ with $\mathbf{x}_{\text{oos}} \notin X$. This is typically referred to the "out-of-sample" extension [Strange and Zwiggelaar, 2014, Sec. 5.2]. The inverse from latent space to ambient space is referred to as "pre-image" [Strange and Zwiggelaar, 2014, Sec. 5.3].

A typical default for the out-of-sample mapping is the Nyström extension described by Eq. 2.43 – 2.42. The extension does not introduce additional parameters and instead only requires the same kernel evaluations to be applied for new samples $\mathbf{x}_{\text{oos}}$ with respect to the available data X (Eq. 2.43). With an abuse of notation the *vectors* $K_{\text{oos}}^{(\alpha)}$ and M_{oos} have capital letters (which are used for matrices) to highlight the connection to the respective previous operation. New samples are required to be in the vicinity of the available point cloud — if a point is an extreme outlier, this can result into a zero vector M_{oos}.

The equation in Eq. 2.42 then describes a function representation of the extended eigenvectors. The extension follows from the relation to the eigenproblem in Eq. 2.40. Note that eigenfunctions $\psi_p(\mathbf{x})$ with associate eigenvalue $\omega_p \approx 0$ are unstable and are usually truncated.

Statement 8: The final step constitutes the diffusion mapping $\mathbf{g}_{\text{dmap}}(\mathbf{x}$. The parameter t (positive value defaulting to zero) corresponds to the diffusion time of the process and induces a family of mappings that describe the geometry at different scales [Coifman and Lafon, 2006b]. The function basis $\mathbf{g}_{\text{dmap}}$ contains P approximated eigenfunctions of the Laplace-Beltrami operator, $\psi_p : \mathbb{R}^N \to \mathbb{R}$.

Since DMAP is an unsupervised machine learning method, there is no objective function that allows different parametrizations (e.g. kernel bandwidth) to be compared directly. This also prohibits cross-validation methods, which only become accessible if the data transformation is integrated in a supervised task [Belkin et al., 2009; Strange and Zwiggelaar, 2014]. Later this corresponds to the system identification.

The Nyström out-of-sample extension is the typical default in the DMAP framework. However, there are also a separate series of publications to improve the method. Often these methods use multiple kernels to better model the manifold [e.g. Coifman and Lafon, 2006a; Fernández et al., 2020; Rabin and Coifman, 2012]. Nevertheless, the Nyström extension often produces accurate and competitive mappings; see Chiavazzo et al. [2014] for a comparison of methods.

2.4.3 Time delay embedding

The previous two sections set the basis to extract geometric coordinates from static data. This section continues to highlight important aspects of geometric considerations of time series data connecting to the state space manifold of a dynamical system. A state has a spatial neighborhood but is also an element of a trajectory that is induced by the system's flow (cf. Fig. 2.1).

An important issue in system identification is that time series states often contain insufficient temporal information to perform a regression from one state to the next, i.e. $F_{\Delta t}(\mathbf{x}_j) = \mathbf{x}_{j+1}$ is not well-defined. This problem is also highlighted as one of the main challenges of system identification in Section 2.2.2. This can be a result from a poor selection of state quantities — favoring spatially relevant quantities over temporal ones — but also limitations in the measurement capabilities or the overall high complexity of the system.

In the geometric perspective, the time series data provide evidence to the underlying state space manifold. The measurement quantities form a geometry that can be seen as a *projection* from the true state space to the ambient data space. In the event of non-Markovian dynamics in the measured states, the projection only provides a *partial* view of the true state space manifold [Deyle and Sugihara, 2011; Dietrich et al., 2016; Kamb et al., 2020]. As a result, trajectories that do not cross on the true state space manifold, now cross in the data projection, leading to ambiguous dynamics at intersection points [Deyle and Sugihara, 2011]. This is illustrated for a single trajectory in Fig. 2.8. After the projection to the measurement space in step 1, the trajectory includes an intersection. Even if the measurement state has a higher dimension than the true state space the intersection persists. The only way to resolve the intersection is to include new temporally informative state quantities.

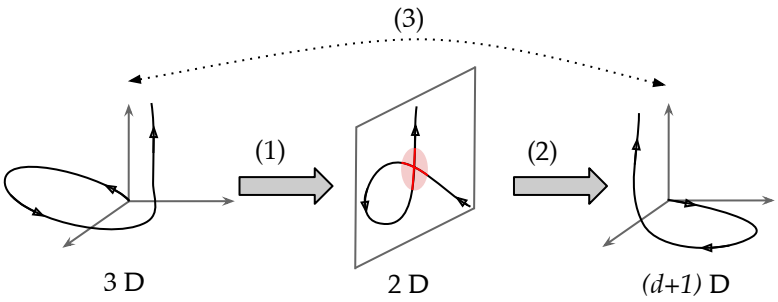


Figure 2.8: A schematic illustration of the effect of time delay embedding. The left plot displays the true system trajectory, which is projected into the measurement space (1). The highlighted area (red) around the intersection is a "false neighborhood" region. In the next step (2) the time delay embedding into a $(d+1)$-dimensional space is performed, where the intersection resolves. The dashed connection (3) highlights a one-to-one mapping between the original and reconstructed trajectory.

The actual point cloud — formed by the time series data — may not contain the *exact* intersection point of a trajectory. However, as pointed out in the previous sections, data-driven methods like manifold learning often rely on pairwise distance measures. Consequently, the points near to the intersection create *false neighborhood* relations. This means that point pairs appear as short-distanced in the projection, while they are far-distanced on the true state space (see Fig. 2.8).

In contrast to static point clouds which can also suffer from partial view projections, for time series data the temporal context becomes beneficial. One way to include temporal information is to estimate the time derivative. However, this only approximates instantaneous change and does not include time information on larger scales and is also sensitive to noise.

An established approach in time series modeling are "state space reconstruction" methods, which augment the measurement states from the temporal context in "time windows" [Sauer et al., 1991; Takens, 1981]. Moreover, the methods are robust to noise and can reconstruct dynamics on larger time scales [Deyle and Sugihara, 2011; Mezić and Banaszuk, 2004; Sauer et al., 1991]. This is in contrast to instantaneous change in a time derivative of a state. A classical state space reconstruction method is the time delay embedding, which is first expressed in its equation form:

$$\mathbf{g}_{\text{td}}(\mathbf{x}_j; d, \kappa) = [\mathbf{x}_j, e^{-\kappa}\mathbf{x}_{j-1}, e^{-2\kappa}\mathbf{x}_{j-2}, \ldots, e^{-d\kappa}\mathbf{x}_{j-d}] = \mathbf{y}_j. \tag{2.45}$$

The embedding $\mathbf{g}_{\text{td}}$ augments a state $\mathbf{x}_j$ with d prior samples of the time series. Each delayed state in the embedding is also weighted with a factor that exponentially decays the of delay $(1, \ldots, d)$ and parametrized with $\kappa \geq 0$. The semicolon denotes a vertical stacking of the state column vectors, leading to a final time-delayed vector $\mathbf{y} \in \mathbb{R}^{N(d+1)}$. Note that the embedding cannot be performed on the first d states of a time series.

Intuitively, the time delay embedding *stretches* point samples apart that are in a false neighborhood relation. In a successful reconstruction this ultimately resolves the intersection points and the embedded states become Markovian [Deyle and Sugihara, 2011].

The theoretical foundation of time delay embedding lies in theorems of Whitney and Takens [1981] which are valid for the *attractor* of a dynamical system [Sauer et al., 1991]. The theory states that including a sufficient number of delayed states (d) in the embedding, reconstructs a space that is topologically equivalent to the original state space manifold. In the theory it is "sufficient" if the delay is set to $d \geq n + 1$ if the compact manifold is of dimension n [Sauer et al., 1991]. The "equivalence" means, that a smooth and invertible one-to-one mapping between the two spaces exists. In mathematical terms this is referred to as *diffeomorphism* [Deyle and Sugihara, 2011; Takens, 1981].

According to the embedding, Eq. 2.45 induces a (reconstructed) manifold that relates to the hidden system's state space manifold (given sufficiently many delays d). Fig. 2.8 exemplifies this in the second step where the projected trajectory is embedded in the delayed space of dimension $d + 1$ (original state plus d delays). After the embedding, the intersection-free trajectory in the right image has the same structural qualities as the original trajectory, denoted by (3) in the figure to highlight the diffeomorphism.

The additional weight parameter κ in Eq. 2.45 is for regularization and a projection onto a dynamically relevant space (the Oseledets space); for details see Berry et al. [2013]; Dietrich [2017].

The foundation of the embedding theorems of Whitney and Takens have led to numerous extensions and theoretical research, and has become an established procedure in system identification. While the original works focus on univariate time series, the theory has been extended to multivariate cases in Deyle and Sugihara [2011] (which is captured in Eq. 2.45). The assumption here is that all state quantities in the time series origin from the same underlying data-generating system. Lagged states are also often implicitly included in statistics-based methods, which model a time series in terms of a auto-regression (cf. Eq. 2.12). The next section provides research links that connect time delay embedding to the Koopman operator.

2.4.4 Research links to system identification and Koopman operator

As introduced in the general form of system identification in Eq. 2.45, finding a suitable (intrinsic) state representation is a vital task. The previous two sections introduced a nonlinear projection to the eigenfunctions of the Laplace-Beltrami operator with DMAP and time delay embedding, as rich transformations to extract spatial and temporal features from data. Each method is based on a strong theoretical foundation. In this section I highlight research that combines the methods to a single state representation and also connects it to the Koopman operator theory of Section 2.3.1.

An interesting and often applied combination is when time delay embedding is performed on time series data *before* it is passed to DMAP. This can be understood to first augment necessary temporal information and subsequently remove redundant spatial information from the states. In a geometric perspective, the time delay embedding induces a new geometry relating to the state space manifold, the subsequent DMAP then extracts relevant and geometric aligned coordinates. The transformation can be expressed in terms of a composed function:

$$\begin{aligned} \mathbf{g}(\mathbf{x}) &= [\mathbf{g}_{\text{dmap}} \circ \mathbf{g}_{\text{td}}](\mathbf{x}) \\ &= [\omega_1^t \psi_1(\mathbf{y}); \omega_2^t \psi_2(\mathbf{y}); \dots; \omega_P^t \psi_P(\mathbf{y})] = \mathbf{z} \in \mathbb{R}^P \end{aligned} \tag{2.46}$$

where $\mathbf{g}_{\text{td}}(\mathbf{x}) = \mathbf{y}$ are the time-delayed coordinates from Eq. 2.45 and $\mathbf{g}_{\text{dmap}}(\mathbf{y}) = \mathbf{z}$ the evaluated eigenfunctions of the Laplace-Beltrami operator, approximated with DMAP in Eq. 2.44. To perform the composed transformation, the states are elements of a time series collection, $\mathbf{x} := \mathbf{x}_j^{(i)}$ (as per Eq. 2.24 for the EDMD framework). When the time delay embedding is considered part of the kernel, then this can be interpreted as a "dynamically-adapted kernel" [Giannakis, 2015].

The following two subsections first link time delay embedding to system identification and Koopman operator theory and then in its composed transformation with DMAP.

2.4.4.1 Research highlighting time delay embedding

Performing time delay embedding on a single coherent time series leads to a special matrix form, which is termed "Hankel matrix". This matrix and its structure is often central to the analysis of time series data. As such it was integrated early in the system identification method Eigensystem Realization Algorithm [Juang and Pappa, 1985].

A downside of time delay embedding is that is typically results in a much higher dimensional state, depending on the original state dimension and the number of delays. Despite its intended use to enrich the state with temporal context, the lagged quantities are usually highly correlated. A natural follow-up operation to the embedding was therefore to perform a truncated SVD on the lagged states. This has led to well-established methods originating from the Singular Spectrum Analysis (SSA) [Broomhead and King, 1986], providing a way to extract principal spatio-temporal features from incomplete measurements and truncate coordinates that relate to noise.

Time delay embedding has also played a vital role in the revived interest in the Koopman operator. Mezić and Banaszuk [2004] describe an analytic method to analyze the asymptotic behavior of dynamical systems via the Koopman operator by using a "statistical Takens theorem". Given that many data settings make state space reconstruction necessary, time delay embedding has also been integrated in a variety of numerical schemes to approximate the Koopman operator. The standard DMD of Schmid [2010] is extended in Tu et al. [2014] to overcome a major limitation of the original algorithm, in which it was not possible to extract the dynamics of a "standing sine wave" as a simple form of a dynamical system. Follow up research describes increasingly complex frameworks, such as the Hankel Alternative View of Koopman (HAVOK) [Brunton et al., 2017] and "Higher-Order" DMD [Le Clainche et al., 2017]. A particularly relevant variation is the Hankel-DMD, described in Arbabi and Mezić [2017]. This study contains a remarkable proof that applying DMD on the Hankel matrix in the limit of infinite delays computes the true Koopman eigenpairs for ergodic systems. In other words, under the given system assumptions, the proof states that increasing the number of delays in the embedding *linearizes* the system dynamics. This is exactly what is required within the EDMD to approximate the Koopman operator.

2.4.4.2 Research highlighting the composed function

While the above research computes a linear truncated basis of time delay states (such as in SSA), the composed transformation in Eq. 2.46 generalizes this aspect to a "geometric-aware" basis. This has led to the "natural extension" Nonlinear Singular Spectrum Analysis (NLSA), where DMAP is integral with a different dynamics-adapted kernel [Giannakis and Majda, 2013]. The authors highlight the analytical power for time series data.

A further study in which the composed function is integral, is the Diffusion-Mapped Delay Coordinates (DMDC) framework [Berry et al., 2013]. The authors give an interesting theoretical background of the transformation and highlight that DMAP performs a globally consistent and best-preserved geometry of the point cloud that is formed by

the time-delayed states. The approximated Laplace-Beltrami eigenfunctions $\psi(.)$ are time series and give additional dynamical interpretation. Because of the order in the eigenvalues and the harmonic coordinates, the transformation performs a time scale separation in slow and fast frequency functions defined on the manifold. That is, the dynamics are sliced into Fourier-like components, where the eigenvalues relate to the time scale. While the research mostly aims at gaining insight into and describing the system properties, Berry et al. [2015] and Dietrich et al. [2016] use these manifold coordinates to directly describe the dynamics on these.

The composed function has then also been integrated into the Koopman operator research branch. Das and Giannakis [2019] provide a connection between the three seemingly different branches of research: attractor reconstruction (Takens), manifold learning (Laplace) and mode decomposition (Koopman). In particular, Giannakis [2019] shows that for ergodic systems and for infinitely many delays ($d \to \infty$) the Laplace-Beltrami operator in Eq. 2.46 commutes with the Koopman *generator*. From this property follows that the Laplace-Beltrami operator shares the same eigenspace to the Koopman operator. The proposed method is not stated in the EDMD framework but has parallels in the numerical treatment. Moreover, the authors use a variant of DMAP with variable kernel bandwidth which generalizes better to non-compact manifolds [Berry and Sauer, 2016]. Based on the results of Giannakis [2019], Kamb et al. [2020] describe so-called convolutional coordinates, which refer to generic projections of time-delayed coordinates onto an orthonormal basis to linearize the dynamics and approximate the Koopman operator.

Justified by these findings and connections, there are a variety of numerical schemes available that integrate DMAP to perform Koopman operator-based system identification tasks. Harlim and Yang [2018] describe a "diffusion forecast" in which the basis functions computed by the DMAP algorithm are used to perform probabilistic predictions. Alexander and Giannakis [2020] propose a framework for "kernel analog forecasting" to forecast nonlinear time series, which also facilitates the analysis of generalization error and quantification of uncertainty. Mauroy et al. [2020, Ch. 14] use the data transformation to describe a "geometrically informed" filtering, analogous to the Kalman filter.

2.5 Software for system identification

So far I have mainly described the methodological part of system identification, with a special focus on operator-based approaches and their numerical approximation. A central element to make use of these theories and associated data-driven algorithms in applications is software. Scientific software is an effective way to express, communicate and replicate results [Goble, 2014]. In modern computationally-centered modeling endeavors, software should be available, discoverable, usable and adaptable to conduct new research [Anzt et al., 2020; Stodden and Miguez, 2014]. Efficient algorithms and well-designed implementations thereof are a key element for data-driven modeling to scale with larger datasets and perform "algorithmic experiments". Software that meets

these requirements is termed "sustainable software" and has become a research asset on its own [Anzt et al., 2020].

Software for data-driven modeling differs from more traditional software engineering approaches in that the exact algorithm to address a problem is not known beforehand. Instead the final software model has data as an additional and integral part. This is a paradigm shift, in which testing and verification of a software becomes more challenging [Khomh et al., 2018]. A typical approach is to make use of well-understood "academic systems" (e.g. the swiss-roll in the previous section) or use widely adopted datasets in a community (e.g. the MNIST dataset, containing handwritten digits [LeCun and Cortes, 2010]). To my knowledge there is no such "standard dataset" for time series data.

Actively maintained scientific software requires a high degree of flexibility and modularity. The field of machine learning is subject to ongoing research, where scientists frequently extend the problem set of data-driven models. Moreover, algorithmic research means that there are often many algorithms available to address the same problem. Because of the different goals to speed up, increase numerical stability or improve model convergence, algorithms often have their strengths and weaknesses for different settings. For example, depending on the dataset size and computational restrictions, a modeler has to decide whether to use an exact algorithm or a fast approximate version.

Section 2.5.1 describes the general interaction points for researchers and modelers for system identification tasks. Furthermore, Section 2.5.1 highlights relevant and popular scientific software, with a focus on the Python programming language and its ecosystem for scientific computing [Harris et al., 2020; Virtanen et al., 2020].

2.5.1 System identification loop

A main source that mandates software flexibility is the general "model selection" workflow for machine learning tasks. The best suited model structure and parametrization for a problem are unknown. The "art of modeling" requires skills and specialized knowledge to manipulate the model and feed back new understanding to iteratively refine and simplify a model [Pintelon and Schoukens, 2012]. A scientific software for data-driven modeling has the requirement to provide an expressive interface that can cover a wide range of structurally different model specifications, without the necessity for a modeler to look at or manipulate implementation details.

In practice the model selection is often a trial and error procedure, reflecting an optimization in a model space. Fig. 2.9 gives an overview of these iterative refinements at different hierarchical levels of the modeling process in a "system identification loop" [Nelles, 2020]. Each step of the loop requires interaction of either the modeler or can be part of an automatized optimization routine [Pintelon and Schoukens, 2012]. Typically, model specifications in the first three steps require prior knowledge depending on application-specific criteria. For the last two steps, generic routines and good practice guidelines are often available that align the model in a best way to the data.

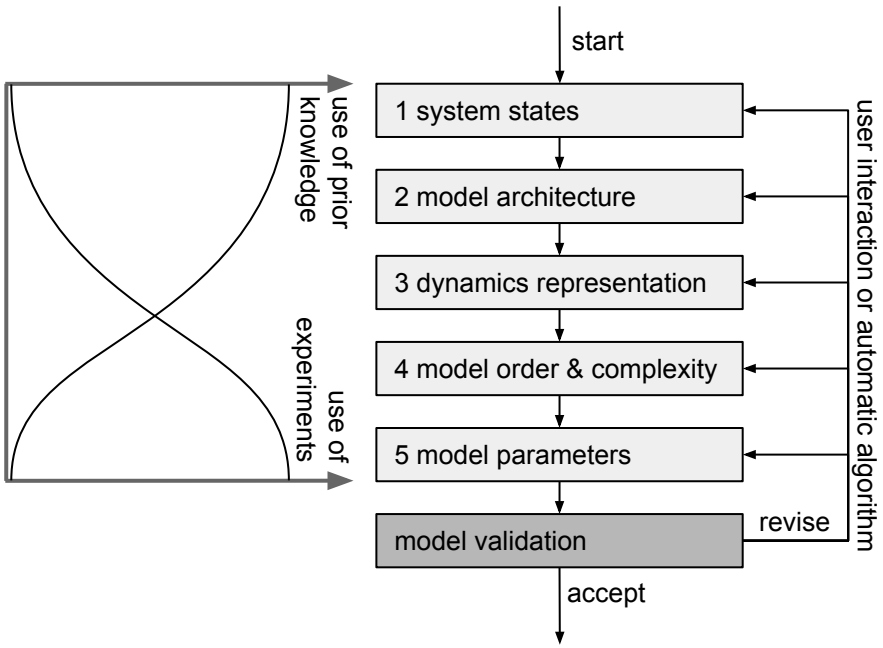


Figure 2.9: The system identification loop, from [Nelles, 2020, Sec. 1.3]. The first five steps are statements inside loop and the last box corresponds to the loop condition. The original source of the figure also covers the case when the system has additional control input.

At the start of the system identification loop is the **model state**, corresponding to the observed measurements of a dynamical system. As highlighted in the previous section, the selection of measurements is critical and should be guided by both spatially and temporally relevant information. This also includes finding and generating a suitable state representation from a fixed set of measurement quantities. This can be linear dimension reduction methods or methods that extract geometric coordinates as covered in the previous section.

While for (isolated) physical processes the measurement state is often obvious, for increasingly complex and multi-modal systems the influence of a quantity on the overall system is less understood — especially if the model state dimension is high [Nelles, 2020]. This can make revising the model state necessary to include new explanatory quantities or drop unnecessary ones. Methods for ranking the importance of state quantities can help. Moreover, there are often quantities that are known to influence a system but cannot be measured. For example, a traffic scenario is influenced by "human factors", which includes the psychology of decision making [Kleinmeier, 2021]. Such factors are unobserved and manifest as noise in the state observations.

In the second step, a user specifies the overall **model architecture**. Section 2.2.3 provides a selection of model architectures for system identification coming from diverse directions, of which the operator-based approach is central in the context of this thesis. The choice of an architecture is usually influenced by technical aspects such as the intended use, accuracy requirements or computational demands. But non-technical

factors such as the experience of a user and the availability of software are also relevant [Nelles, 2020].

Within the scope of the selected model architecture, in the third step a modeler can then specify a **dynamical representation**. Besides the temporal evolution of the model state, this can also include other dynamically relevant quantities such as the uncertainty of a prediction [e.g. Alexander and Giannakis, 2020]. A main distinction in the representation is between the system's (continuous) vector field or its (discrete) flow (Eq. 2.1 – 2.2). Both representations can be independent of time (autonomous) or change with time (non-autonomous). For the special case in which the dynamics are in a linear form — as in the Koopman operator framework — this also permits a *spectral* representation of the dynamics.

The fourth step requires a modeler to specify the **model order and complexity**. This is a vital task in data-driven modeling and typically formalized by balancing the bias-variance trade-off [Murphy, 2012, Sec. 6.4.4]. The model can be adapted to steer the complexity and match the model with the available information in the data. Suboptimal models either fail to capture principal patterns in the data (high bias, low variance) or overfit the data by describing high order and noise-corrupted patterns (low bias, high variance). Non-parametric models have no definite complexity order because they can arbitrarily match the training data by including new parameters.

The last step of the system identification loop describes setting the **model parameters**, which are introduced along the previous steps of the loop. The parameters explicitly specify the internal learning algorithm and functions. Altogether the parameters can interact and influence different model characteristics, such as the convergence rate, numerical stability or prediction accuracy. To reduce the risk of overfitting it is important to include parameters that regularize the model's complexity. Fortunately, this last step is the easiest to automate, because it mainly requires data to quantify what parameter settings are better suited. In machine learning there are established procedures for parameter optimization. A limiting factor here is the build time to fit a new model, because faster build times permit to sample the parameter space more thoroughly.

Finally, as the loop condition, the **model validation** describes criteria that must be fulfilled such that the model is suitable for its intended use. These criteria are problem-specific and include all model specifications of the preceding steps [Nelles, 2020]. For the common case of a predictive model, the validation includes evaluating the forecasting error between the model and true system observation over a time horizon. For practical relevance the validation should be performed on data that is excluded from the model construction process [Nelles, 2020]. Otherwise the validation is highly likely to be too optimistic about the performance, because the model is biased towards known data. Moreover, the validation can provide valuable qualitative insights to revise specifications that are often not possible to automatize. For example, systematic errors can indicate missing explanatory features in the model states, while errors that bias over time promote model architectures that support updating a model when new data becomes available [Hemati et al., 2014]. Note that many common practices for model

validation for static data are not transferable to time-dependent settings [Bergmeir and Benítez, 2012].

2.5.2 Python scientific computing stack for machine learning

Covering the system identification loop requires a high degree of flexibility in scientific software. However, the flexibility should not only be restricted to an isolated research project, but also embed to a software environment in a wider sense. Typically, re-implementing functionality of which good solutions exist should be avoided [Anzt et al., 2020]. Including existing code then often leads to complex and large software dependencies [Ma et al., 2016]. The degree of how influential a software is, can be measured by how many follow-up projects make use of it [Ma et al., 2016]. This section gives an overview of the popular scientific programming language Python [VanRossum and Drake, 2010] and related software packages that relate to system identification tasks.

Python is an interpreted, platform-independent, object-oriented, and dynamically typed programming language [VanRossum and Drake, 2010]. It addresses a large target audience with a growing community of scientists, engineers and practitioners of diverse disciplines. It has become popular for a variety of applications, such as web development, system tools or scientific computing [Ma et al., 2016]. Importantly, because of its features it has become the primary language for machine learning tasks [Gonzalez et al., 2020, Fig. 6]. The quality as a general purpose language is a distinguishing factor to other "scientific programming languages" that have a more specific target audience; Table 2.5. The advent of open-source movement — as a particular important factor for data-driven modeling [Sonnenburg et al., 2007] — is a reason of why many of the scientific community have moved away from proprietary software [Paszke et al., 2019].

From the start the Python programming language emphasized readability and expressiveness. It is, for example, possible to exchange any object (attribute, function and class) easily during runtime without specific type requirements and class hierarchies. This favors rapid prototyping and has led Python to become one of the standard languages for exploratory algorithmic experimentation and modeling [Millman and Aivazis, 2011]. Specific software patterns applied in this context make use of this flexibility, which even allows separately developed software packages to become interoperable without requiring any explicit dependency. A main pattern to achieve this, is the so-called "duck typing" principle, which stems from the explanatory phrase: "If it walks like a duck and it quacks like a duck, then it must be a duck". The meaning behind is, that objects are identified by their interface instead of their actual object type. If an object provides the necessary attributes it can be passed to external software without further manipulation or hard dependencies. In contrast, statically-typed languages can only achieve a similar flexibility by introducing increasingly abstract and complex class hierarchies in software architectures.

Table 2.5: Overview of prominent interpreted programming languages used for scientific tasks.

Name	Target audience	Availability	Est.	Web
Matlab	scientific computing and engineering	proprietary	1984	mathworks.com
Julia	scientific computing and engineering	open-source	2012	julialang.org
R	statistical modeling and analysis	open-source	1993	r-project.org
Python	general purpose	open-source	1991	python.org

All third-party software in Python is organized in packages and available in the official index PyPI[2]. Many popular packages are organized by communities, which despite being third-party to Python, provide quasi-standard packages for the software ecosystem [Harris et al., 2020; Virtanen et al., 2020]. For scientific computing and data-driven modeling the quasi-standard packages are part of the Scientific Python (SciPy) ecosystem, which is managed by NumFOCUS [Harris et al., 2020]. SciPy promotes open-source Python-interfaced software for mathematics, science and engineering with high-quality standards in terms of availability, documentation, testing and a culture that strives to improve the community-driven software development and management [Millman and Aivazis, 2011; Virtanen et al., 2020]. This has also attracted many other software projects, which align their interfaces to the packages that are included in SciPy [Gonzalez et al., 2020]. For example, large technology companies have promoted software projects to efficiently design and use deep neural network architectures, such as *TensorFlow* (Google, Abadi et al. [2016]) or *PyTorch* (Facebook, Paszke et al. [2019]).

The flexibility of the Python language, however, also has drawbacks, such as a reduced execution speed and reduced capabilities to detect type errors compared to compiled languages. A main reason for these drawbacks is the code interpretation during runtime, which includes many attribute checks. This is in contrast to compiled languages suitable for high performance applications, such as C or C++. Another restriction is that the Python interpreter process is locked with a so-called Global Interpreter Lock (GIL), which reduces the capabilities of parallel code execution. This means shared-memory parallelization is mostly not possible from within pure Python code [VanRossum and Drake, 2010].

However, for SciPy-applications that heavily rely on efficient (parallel) code execution these restrictions turn out to be only a minor issue. This is because in SciPy a two (or more) language paradigm has been established, where computationally demanding parts are outsourced to low-level and compiled code (usually C, C++ or Fortran) [Dubois, 2007; Oliphant, 2007; Virtanen et al., 2020]. These low-level code executions release the GIL and all of its restrictions. The advantage of this paradigm is that Python provides a high-level and flexible programming front-end which is suitable for scien-

[2] https://pypi.org/

tists and modelers to steer the data and computation flow, without the need to consider low-level implementation details such as memory management [Dubois, 2007]. If there is no low-level implementation of an algorithm available within Python it is also possible to integrate own low-level code or bridge to existing software projects of compiled languages [Behnel et al., 2011].

Overall, the two language paradigm aims to integrate the "best of both worlds". This has immensely decreased the development time for data-driven applications. This allows providing complex processing pipelines to be specified at multiple interaction points (Fig. 2.10) and at the same time execute computationally expensive software parts in compiled languages.

Implementing software in this setting often obscures the underlying computational complexity and has also led to a dedicated set of good practice and programming styles in Python. For example, it should be avoided to reduce the number of costly context switches between interpreted and low-level language.

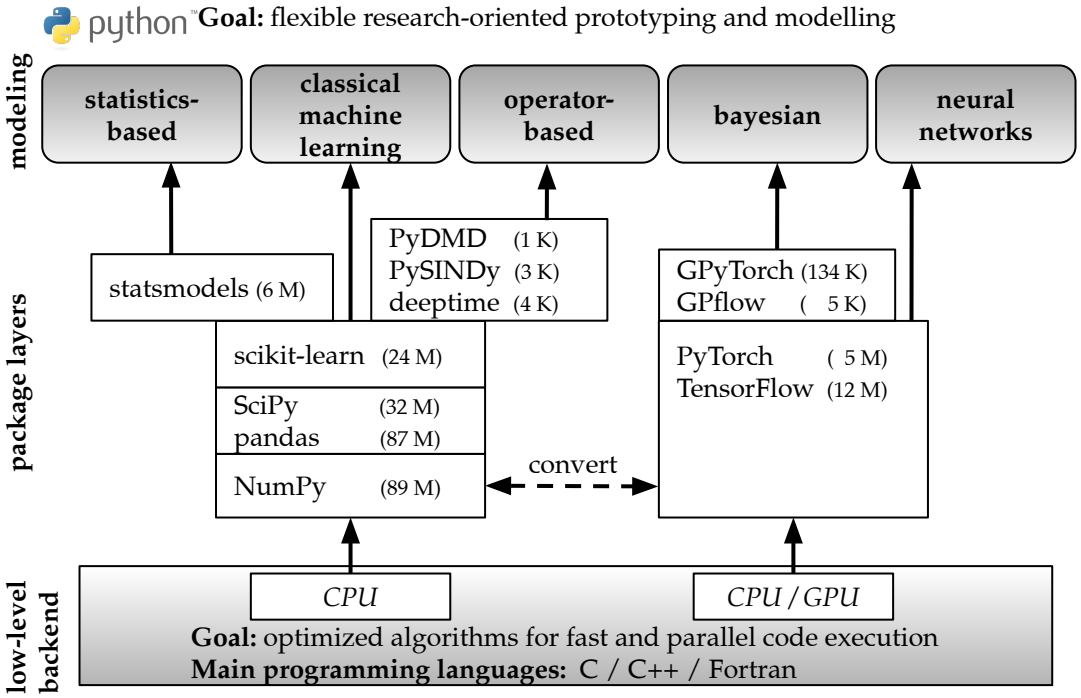


Figure 2.10: Representative Python packages for different modeling approaches involved in system identification tasks (Section 2.2.3). The hierarchical order represents the multi language paradigm of SciPy. The numbers in brackets highlight how often a package was downloaded in September 2021 (M=million, K=thousand). The download statistics are retrieved from the two main public package managers PyPI and conda. Note that there are also dependencies between the packages.

Fig. 2.10 displays open-source packages in the Python ecosystem which are commonly involved for data-driven modeling and can be used for system identification. This does not represent an exhaustive list and there is a diverse range of packages that provide alternative implementations. The middle package layer show two software structures (blocks), leading to the dedicated modeling approaches, as covered in Section 2.2.3. The connections only describe the main modeling objective of a package. There are also cross-connections, for example, *scikit-learn* also provides an implementation of a Gaussian Process. The tendency is that the closer a package is to modeling end, the more alternative packages or implementations there are. While there are well-defined application programming interfaces (APIs) for data-driven modeling of static data, there are no such standards for the special task of system identification available.

The left block contains elements of the typical "scientific computing stack" in Python. The packages in the stack are open, meaning that higher-level projects also often bind to low-level backend code. The other right block includes software that have evolved with the rise of DNN in machine learning. The separated software from the "traditional stack" includes new frameworks, which are dominantly used to specify a diverse range of DNN model architectures. Because the training of such models is also computationally demanding, these packages typically favor dedicated hardware, such as graphical processing units (GPUs) to train a model.

- ***NumPy***

  The package is the foundation of scientific computing in Python [Harris et al., 2020]. It provides elementary and extensible structures to access, manipulate and operate on data in multidimensional arrays (vector, matrix or tensor). The main data structure is the `ndarray` which captures data of identical type and size (typically numerical types) in row-contiguous memory. The view-based in-memory model tries to avoid copies of data whenever possible and targets computations on CPU. *NumPy's* backend is mainly written in C and provides access to basic linear algebra routines through bindings to Basic Linear Algebra Subroutines (BLAS) and Linear Algebra PACKage (LAPACK).

- ***SciPy, pandas***

  The two packages are essential for scientific computing in Python and tightly couple with *NumPy*. The SciPy package — giving the name to the overall SciPy community — provides a large range of fundamental numerical algorithms that appear across many scientific disciplines [Virtanen et al., 2020]. This includes sparse matrix data structures, interpolation, optimization routines, solvers for ordinary differential equations. Moreover, it interfaces to many well-established backends, such as ARnoldi PACKage (*ARPACK*) for iteratively solving an eigenproblem.

  The *pandas* package extends the *NumPy* data structure to provide a solid foundation for large scale data analysis [McKinney, 2011]. The most prominent data structure is the `DataFrame`, which stores data in a two-dimensional tabular format. Internally, a `DataFrame` is managed in column-oriented *NumPy* arrays. The key extensions

compared to a standard array are that each data point has an associated index (both row and column) and that a single `DataFrame` can have a dedicated data type per column. Moreover, the package provides a rich set of operations that act on the data structures, which includes aggregating, grouping, input-output and statistical operations.

- ***scikit-learn***

  The package [Pedregosa et al., 2011] mainly builds on the *NumPy* and *SciPy* library to provide a set of well-established and diverse machine learning algorithms. The package is part of the "scikit" family, which are domain-specific packages complementing the more general scientific computing ecosystem [Buitinck et al., 2013]. The API design operationalizes the common machine learning concepts such as supervised and unsupervised tasks, but also related procedures like model selection with cross-validation [Buitinck et al., 2013]. The algorithms in *scikit-learn* assume time-independent data, where the data input are typically unlabeled two-dimensional *NumPy* arrays.

- ***statsmodels, PyDMD, PySINDY, deeptime***

  These four packages are examples of specialized data-driven modeling. All packages have dependencies to the fundamental scientific computing stack and mimic the API from *scikit-learn*. The package *statmodels* [Seabold and Perktold, 2010] provides an advanced and specific set of algorithms for statistical data analysis. For the particular case of time series modeling this includes a variety of state-space models and auto-regressive algorithms (e.g. Auto-Regressive Moving Average and extensions).

  The other three packages have been established recently and relate to the operator-theoretic approach. The package *PyDMD* [Demo et al., 2018] includes a variety of DMD algorithms. *PySINDy* [de Silva et al., 2020] mainly focuses on the SINDy approach [Brunton et al., 2016] to find a sparse set of functions that describe a nonlinear dynamical system. *Deeptime* [Hoffmann et al., 2021] is a package for time series analysis, which includes an implementation of the EDMD.

- ***TensorFlow, PyTorch, GPflow, GPyTorch***

  The two packages *TensorFlow* [Abadi et al., 2016] and *PyTorch* [Paszke et al., 2019] include machine learning frameworks with a focus on constructing and training DNN architectures. The target audience ranges from scientists that require rapid research-driven experimentation to practitioners who intend to use the software in industry applications [Abadi et al., 2016]. Paszke et al. [2019] outlines three important trends — array-based programming, automatic differentiation and community-driven open-source software — that have converged most deep learning frameworks to provide a Python interface. Because *NumPy* mainly addresses CPU computations, the new low-level implementations filled a gap in Python's ecosystem with new array implementations and operations that also run on distributed or GPU hardware [Harris et al., 2020]. However, both frameworks also provide an API that aligns with the traditional stack (currently experimental in *TensorFlow*).

Because the DNN software frameworks are designed to be flexible and extensible with custom (differential) functions or operations, this has led to a variety of projects that build on the frameworks. Fig. 2.10 includes two projects — *GPflow* [Matthews et al., 2017] and *GPyTorch* [Gardner et al., 2018] — that implement kernel-based GPs. These packages can directly profit from efficient computations and automatic differentiation.

All of the highlighted projects are examples of a sustainable software development, which includes a variety of additional factors beyond the actual source code, such as community management, documentation or testing [Anzt et al., 2020]. A well-maintained project can promote an easy spread of the underlying methodologies, which is showcased by the deep learning community [Paszke et al., 2019]. The increasingly high complexity of algorithms and the many interaction points in the modeling process, often make scientific software a community effort [Sonnenburg et al., 2007].

The next chapter continues by describing a new software, developed in the course of this thesis. The main focus lies on data-driven algorithms covered in this chapter to approximate the Koopman operator with EDMD and finding suitable state representations by using time delay embedding and manifold learning with DMAP. The proposed software covers the entire system identification loop, from defining necessary data structures to handle time series collection data to optimizing parameters in a grid.

Despite a large body of scientific research and interest in these mathematical models, when I started my thesis no openly available software fully captured and could combine the methodologies of DMAP and EDMD. The three available packages in this operator context (cf. Fig. 2.10) have since been initiated during my thesis; I use them to compare software features in the next chapter.

2.6 Summary

This first chapter has set up the scientific context for my thesis contribution. At the center are dynamical systems and the identification (i.e. "learning") of these from time series data. There are various modeling approaches, which I classified based on their origin — statistics-based, Bayesian and machine learning. However, I emphasize an operator perspective as a complementing approach that has emerged as a promising candidate for system identification.

The Koopman operator theory describes a canonical dynamical system in a linear albeit infinite-dimensional function basis. A major advantage of the numerical treatment in the generic EDMD framework is that linear regression is sufficient to capture the system's dynamics. The main focus on the methodology lies on finding a suitable state representation in which (1) the dynamics linearize and (2) the original measurements can be (linearly) reconstructed. If such a representation is available, the nonlinear dynamics can be described in a linear dynamical system. This mitigates many of the modeling challenges that arise from identifying nonlinear and high-dimensional sys-

tems directly. Overall, the linear model structure profits from a sound mathematical foundation, which allows characterizing the system into its spectral components — the Koopman triplet (modes, eigenvalues, eigenfunctions).

One way to extract a suitable state representation from empiric data is based on geometric understanding of the data-generating system. In particular, the Laplace-Beltrami operator is a central object to describe and analyze the geometry of data. The eigenfunctions of the operator provide a function basis, in which the elements is oriented on the intrinsic manifold (cf. Fig. 2.6). As a kernel-based method, Diffusion Maps is a widely-adopted numerical framework that is capable of approximating the eigenfunctions of the operator from high-dimensional and unstructured data.

Another data transformation is the time delay embedding. Here the measured states are augmented with the temporal context within time series. The embedding can reconstruct essential parts of the state space manifold such that the new state dynamics become well-defined (cf. Takens [1981] theorem). In a geometric perspective, the embedding "stretches apart" false neighborhood relations, which are typically introduced if the measured state quantities contain insufficient temporal information (cf. Fig. 2.8). All of the methodologies and underlying theories — connected to Koopman, Laplace, Takens — are interrelated: For ergodic systems, the eigenspace of the Laplace-Beltrami operator on infinite-delayed states is the same for the Koopman operator [cf. Giannakis, 2019; Korda and Mezić, 2018].

Making use of the operator-informed approach for system identification, however, requires scientific software. Python has become established as the main programming language for data-driven modeling. In a two-language paradigm it provides an easy syntax that promotes a flexible modeling front end, while making use of efficient algorithms in a low-level and compiled programming language. However, on a software map of available packages, there are no established standards for system identification and no software that would allow integrating the covered data-driven methods to approximate the two main operators in a single processing pipeline. In the next chapter I start with my contribution of the thesis, where I fill this gap by transferring these methodologies to scientific software.

3 Software for operator-informed system identification

In this chapter, I describe the scientific software *datafold* as the first part of my thesis contribution. In *datafold* I transfer the numerical methods of the previous chapter in a structured software design to promote **data**-driven modeling within a mani-**fold** context. The goal is to boost the use of an operator-based modeling approach in a machine learning setting and to enable the concrete data analysis of Chapter 4, which is the second part of my thesis contribution.

Datafold covers the entire system identification loop from Fig. 2.9, where a model can be revised at each of the intersection points within the loop. All methods can therefore be used on their own, serve as building blocks in new methods or be used together in a single data processing pipeline. I confine my descriptions of the software to the main design decisions and ideas. This means that I mostly leave out implementation details, which are available in the source code provided in the Supplementary Material. The software has been published and peer-reviewed in the Journal of Open Source Software (JOSS) in Lehmberg et al. [2020b].

In Section 3.1, I highlight the main target model architecture to perform operator-informed and nonlinear system identification. My goal is to use this architecture to extract geometric and dynamically relevant coordinates from the identified system. I then give an introduction to *datafold* in Section 3.2, which covers the overall software architecture and measures installed to promote a high software quality.

Datafold has three main components, which I describe in a bottom-to-top fashion from Sections 3.3 to 3.5 following the steps of the system identification loop. Note that this reverses the order in which I describe the methods in the previous chapter. The main software contributions can be summarized as:

- a new data structure for time series collections (Section 3.3)
- a new efficient algorithm to compute a sparse δ-range distance matrix (Section 3.3.2)
- methods to extract a suitable state representation from data — with an emphasis on time delay embedding (Section 3.4.2) and Diffusion Maps (Section 3.4.3)
- nonlinear system identification with a generic implementation of Extended Dynamic Mode Decomposition (Section 3.5) including parameter optimization (Section 3.5.2)

3.1 Main operator-informed setting

Here I describe an operator-informed model architecture that consolidates the main methods of the previous chapter. The goal is to build a model to perform nonlinear system identification that also extracts explicit and accessible coordinates that relate to the system's geometry and dynamics. Fig. 3.1 illustrates the overall setting, where each component fulfills a dedicated task. While I designed *datafold* to be flexible to parametrize a wide range of model configurations, I use these main components to highlight the organization within the software in the following sections.

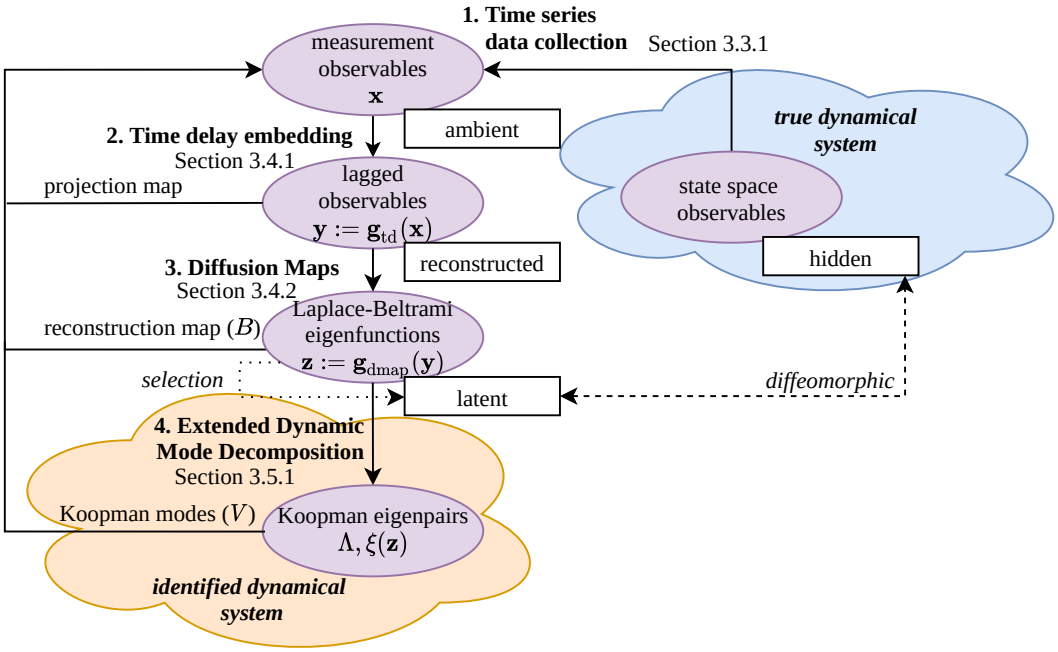


Figure 3.1: Schematic overview of the main setting in this thesis to perform operator-informed system identification and to extract geometric and dynamic coordinates from time series data. The observable spaces are in ellipses and describe a function basis, that could be used as a dictionary within Extended Dynamic Mode Decomposition (EDMD). Each observable space is defined on an associate state space in the attached rectangle. The solid arrows correspond to steps that involve discretization or data-driven approximations. The arrows back to the measurement states describe the reconstruction map ($G(\mathbf{z})$ in the general system identification form in Eq. 2.11 on page 17). The dotted arrow highlights an additional procedure required to obtain the latent space and the dashed arrow corresponds to a one-to-one relation (diffeomorphic) if the reconstruction was successful.

The *true dynamical system* corresponds to the system under study. It captures the exact dynamics in hidden (unknown) state variables. To identify this system, the first step is to collect measurement time series, which are the basis for data-driven model construction. The data collection can be seen as a discretization and projection of the hidden state space into the ambient space. Within the Koopman operator view, the

collected data correspond to measurement observables, $\mathbf{g}_{\text{ID}}(\mathbf{x}) = \mathbf{x}$ (see Eq. 2.20). The standard Dynamic Mode Decomposition (DMD) uses these observables directly to approximate the Koopman operator. However, this assumes linear dynamics, which is often not fulfilled in practice. All follow up processing steps reconstruct these original measurements (see arrows on the left side in Fig. 3.1), to be able to perform forecasts and model validation with the empirical time series of the true dynamical system.

The first two processing steps on the measurement data correspond to the composed data transformation $[\mathbf{g}_{\text{dmap}} \circ \mathbf{g}_{\text{td}}](\mathbf{x})$. The objective is to obtain a new and suitable state representation (Eq. 2.46 on page 50). The time delay embedding augments the states with temporal context to reconstruct partially observed dynamics in the measurement states. These "lagged observables" are, for example, used within the Hankel-DMD [Arbabi and Mezić, 2017]. The subsequent projection onto the eigenfunctions of the Laplace-Beltrami operator forms a geometrically aligned function basis on the (reconstructed) data. The requirements on this intrinsic state representation are high: (1) the dynamics in these states should be (approximately) invariant (i.e. evolve linearly under the dynamics), (2) should be suitable to reconstruct the original measurements, and (3) provide valuable intrinsic system information. Fortunately, as highlighted in Section 2.4.4, this composed function has many favorable (theoretical) qualities such as a geometric interpretation [Berry et al., 2013] and connection to the Koopman operator [Giannakis, 2019]. Moreover, I use the function basis to (4) regularize the model's complexity by choosing how many eigenfunctions to include in a model. The regularization is possible because the Laplace-Beltrami eigenfunctions are sorted by their respective eigenvalues, which connect to the frequency of the eigenfunction [cf. Fig. 2.6 and Berry et al., 2013]. Highly-oscillating functions on the manifold (with small eigenvalues) are likely to capture directions that associate to noise in the data. In my thesis, I explore whether these requirements on the intrinsic states hold in concrete data scenarios in Chapter 4.

Once available, I use the new state representation for two tasks (both of which are highlighted in Section 2.4.2). First, I parametrize a latent state space that is one-to-one to the hidden state space (diffeomorphic in mathematical terms, as highlighted in Fig. 3.1) by selecting a parsimonious and geometrically informative subset of the Laplace-Beltrami eigenfunctions. Secondly, I approximate the Koopman operator for system identification, whereby the Laplace-Beltrami eigenfunctions provide the final (truncated) function basis (vertical direction of Fig. 3.1).

Finally, the identified Koopman operator-based system is decomposed into the spectral components of the Koopman triplet $(V, \Lambda, \xi(\mathbf{z}))$; modes, eigenvalues and eigenfunctions. Together these components define the final model to describe the original observations and perform predictions with a linear dynamical system (see Eq. 2.30 – 2.32 on page 35). Ultimately, these spectral coordinates are accessible for interpretation and provide valuable insight into the identified dynamical system.

In Section 2.4.4, I highlighted research that gives a strong theoretical background to this setting. However, to the best of my knowledge no other research papers (except my own in Lehmberg et al. [2021]) describe the combination of time delay embedding

and Diffusion Maps (DMAP) within the generic EDMD framework (the computations to approximate the Koopman generator are similar in Giannakis [2019]). Moreover, no publicly available software would allow specifying this setting without additional implementation effort. This includes the operator-informed software projects initiated during the time of my thesis. To highlight my distinctive contributions, I compare the features of existing software to *datafold* throughout this chapter.

3.2 Introduction to *datafold*

Before describing the main software components, I first provide a general introduction to *datafold*. In Section 3.2.1, I highlight why it was necessary to start a new software project and why it became an important element for my data analysis. I then outline the main design decisions and hierarchical software architecture in Section 3.2.2; the architecture provides orientation for the detailed sections to follow. Finally, in Section 3.2.3, I highlight measures that I installed to achieve high software quality.

3.2.1 Statement of need

A general rule in computer science is to avoid creating new software for which a well-suited solution is already available (leaving aside licensing issues or educational purposes) [Anzt et al., 2020]. During my thesis, I identified a gap between the active research field of studying linear operators for data-driven modeling and the availability of scientific software making these methods accessible. The "promises" often communicated in research articles, highlighting the advantages of the operator-based research in system identification, such as

> *Data-driven modeling using linear operators has the potential to transform the estimation and control of strongly nonlinear systems* [Mauroy et al., 2020, p. 197]

> *Koopman operator theory has recently emerged as the main candidate for machine learning of dynamical processes* [Mezić, 2020, p. 1].

had not been transferred to the software and application side of research. As highlighted in Section 2.5.2, qualitative scientific software is an essential element in data-driven modeling. It unifies algorithmic innovations within a field and can make a methodology accessible to a broader audience. Moreover, if a software is accepted by a wider community it can accelerate the methodology as a whole by incrementally integrating new results [Buitinck et al., 2013; Paszke et al., 2019].

Currently, modelers and researchers new to the field of operator-based methods are confronted with two main obstacles. First, they must get accustomed to the terminology in the literature. Because the methods often have a strong theoretical foundation (which is certainly a plus from a scientific point of view), articles often include mathematical concepts from different lines of research, such as function analysis, geometry or dynamical systems. These concepts can be hard to comprehend at first (depending

on the researcher's background) [Bakker et al., 2020]. For a "challenging example" that combines the methods of the main model setting of the previous section, I refer to the research article of Giannakis [2019].

If a researcher is convinced to use operator-based methods, the second obstacle is to transfer the mathematical concepts into the software. An error-free and efficient implementation is a challenging task because it requires a numerical and algorithmic understanding of operator theory [Mauroy et al., 2020, Ch. 7]. This is compounded by the fact that there are no standards for data structures or application programming interfaces (APIs) for system identification in the Python ecosystem, as the main programming language for data-driven modeling [Gonzalez et al., 2020]. Existing concepts are mostly incompatible with each other. This contrasts with the case of modeling static data where the standard data structure is given by a *NumPy* array and *scikit-learn* providing a widely-adopted API. Moreover, the practical aspects of software treatment are typically not covered in methodological articles and very few publications follow the good practice of providing source code in the supplementary material. But even if code is available, the code serves merely as a template. This is because the intention is to provide the code to reproduce the results and not to cover classical software engineering aspects that are necessary to provide general purpose and interoperable methods.

I designed *datafold* to be extensible and open for future development. By building off widely-used software I could avoid re-implementations of existing functionality. As a novel contribution, my software covers the complete data-driven workflow, from low-level specifications, such as time series data structures, to high-level tasks for nonlinear system identification and procedures for parameter optimization.

3.2.2 Software architecture and design decisions

A common goal in software engineering is to create strong abstraction boundaries between the software's components to promote a modular design. This helps to create a maintainable software basis in which it is possible to make isolated changes and improvements that do not affect the entire code basis at once [Sculley et al., 2015].

Datafold is divided into three sub-packages, reflecting a three-layered software architecture; Fig. 3.2 gives a visual overview. Each layer is open, in that components can utilize the functionality of the same or any of the previous layers.

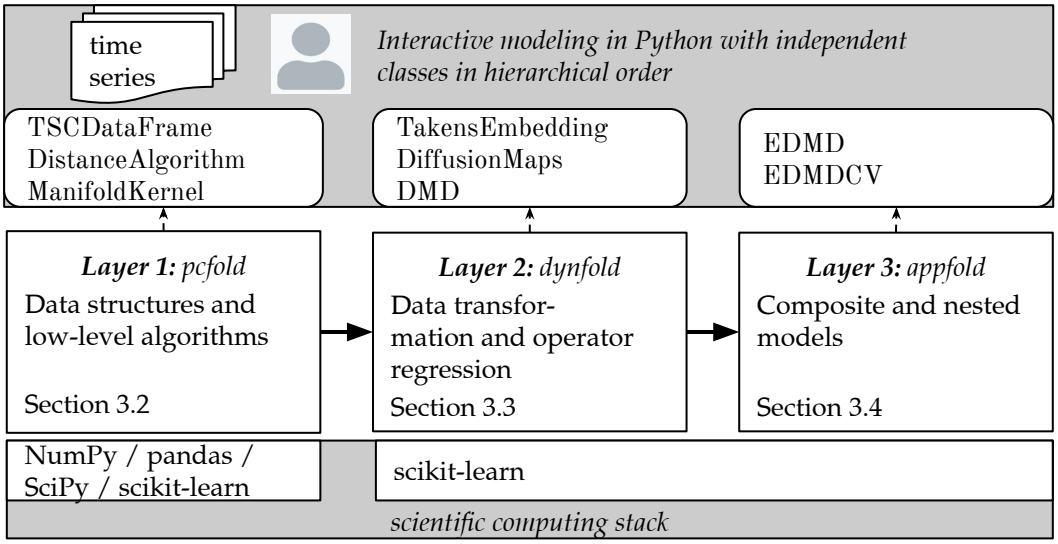


Figure 3.2: Overview of the three-layered architecture of *datafold*. The methods that are in the open layers can be used independently or together in complex model structures. To avoid re-implementations and provide common interfaces, *datafold* is based on the scientific computing stack.

The layered architecture gives a hierarchy that encapsulates important processing steps in the system identification loop in Fig. 2.9. Note that the way I adopt the three-layer pattern differs from the usage of a three-layer pattern. Usually, the highest layer is treated as a "presentation layer" that provides the final interface to the user [e.g. Yokoyama, 2019, Fig. 1.1]. In contrast, the design idea of *datafold* is to separate methods by their complexity: The methods on higher layers are increasingly involved and sophisticated in that they require the functionality from previous layers. A modeler can decide to only use methods provided on the first or second layer for the modeling endeavor. I named each of the sub-packages in the naming style of *datafold*:

- *pcfold* (*pc* refers to point clouds): The lowest package layer includes data structures and basic numerical algorithms. These build on the fundamental packages of the scientific computing stack.
- *dynfold* (*dyn* refers to dynamics): The intermediate level provides (unsupervised) machine learning methods to transform time series to a new and more suitable state representation. Moreover, the layer contains basis DMD variants to perform linear system identification.
- *appfold* (*app* refers to application): The third and highest layer contains methods that solve the complex task of nonlinear system identification, including parameter optimization on cross-validated errors.

The following list summarizes the main design decisions and software requirements for *datafold* and the rationale behind them:

1. **Cover the entire system identification loop**

   Datafold provides a holistic software solution in which it is possible to specify a model at the different interaction points of the system identification loop. For example, within the main model setting described in Section 3.1 it is possible to specify a state and dynamics representation. While the model approach is non-parametric, the complexity can be balanced. Each component has its own set of parameters that can be optimized by automatically evaluating a validation error.

   The flexibility within *datafold* comes at the cost of computational performance. For example, connecting multiple methods in a data-processing pipeline may perform unnecessary data validations and could therefore be implemented more efficiently in a single dedicated method.

2. **Use Python as the main programming language**

   Python has become the main candidate for machine learning tasks [Gonzalez et al., 2020]. As highlighted in Section 2.5.2, the language has favorable features that make it easier to provide an interactive modeling front-end, while steering data to efficient low-level code executions; see Fig. 2.10.

   A feature of Python is that the syntax itself promotes self-documenting and clean code, which is often referred to "Pythonic code". While there are no strict guidelines for what constitutes "clean code", within the programming language there are many established conventions and common idioms for the programming style, such as "explicit is better than implicit". In larger projects, clean code becomes necessary to increase development efficiency and maintainability. An in-depth introduction to Pythonic code is given in Anaya [2021].

3. **Integrate into the scientific computing stack**

   Datafold builds on the Python scientific computing stack to make use of and extend widely-adopted data structures and align to established APIs's. The common packages of the scientific computing stack (left side of Fig. 2.10) provide well-documented, tested, profiled and widely-adopted numerical algorithms and data structures. While the stack includes different packages, these are largely compatible with each other. This combination of a large and organized community, where code is open for inspection, modification and redistribution, provides an excellent code basis. Ultimately, the design decision to align to existing software structures prevents "glue code" as a typical anti-pattern often found in machine learning software. As Sculley et al. [2015, p. 5] describe: *"An important strategy for combating glue code is to wrap black-box packages into common APIs. This allows supporting infrastructure to be more reusable and reduces the cost of changing packages".*

   A drawback of package integration is an increased maintenance in *datafold* to keep up with recent developments in the upstream code. However, because all packages strive for backwards compatibility, this effort is minimal for publicly exposed objects.

4. **Align data-driven methods to *scikit-learn's* API**

   The *scikit-learn* package, as part of the scientific computing stack, defines a quasi-standard API for machine learning in the Python ecosystem [Buitinck et al., 2013; Pedregosa et al., 2011]. While the API focuses on static data, in *datafold* it is necessary to extend the API to process time series data. A great advantage of integrating *scikit-learn* is that it already provides a class organization into typical categories of unsupervised and supervised machine learning tasks by making use of design patterns. For *datafold* I can mirror this organization — building on the experience of a large community [Buitinck et al., 2013] — and adapt it to the workflow for system identification. A further advantage of aligning *datafold's* API to a common machine learning API is that it is familiar to a large audience, which makes it easier for experienced modelers who are new to *datafold*.

   The *scikit-learn* package has already been extended for system identification [de Silva et al., 2020; Demo et al., 2018; Löning and Király, 2020]. However, I found the design decisions in these packages either impractical or too restrictive for the applications that I address later in this thesis (Chapter 4) — I compare these aspects in the next sections.

The following list contains non-requirements, to better highlight the scope of *datafold* within this thesis. However, the first two points are highlighted as future directions at the end of the thesis in Section 5.2.

1. **No streaming setting**

   The nature of time series data is that new samples become available over time. In a streaming environment a model is updated on the fly whenever new data becomes available. This is particularly suitable for systems with changing patterns over time. The model is then capable of adapting to these new patterns. However, most algorithms currently available in the operator-informed setting are described in a batch processing scheme in which all data is readily available. The algorithms for a streaming setting require dedicated adaptations.

2. **No exogenous input or control**

   The Koopman operator framework also extends well to control tasks [e.g. Mauroy et al., 2020]. For the data analysis or algorithms in this thesis, I focus on system identification tasks without any additional exogenous or control input.

3. **No dedicated hardware settings**

   There are sophisticated hardware settings that allow better scaling with the number of samples in a dataset. For example, this includes graphic processing units (GPUs), clusters, or supercomputers. However, addressing multiple hardware configurations restricts the general applicability of the software and complicates the implementation. Therefore, the target hardware setting is a standard CPU setting with potentially multiple kernels and shared memory parallelization. The software therefore works on common general purpose computers.

4. **No graphical user interface**

   Finding and setting up a data-driven model requires great flexibility. The modeling process is therefore best specified with a programming language and not in a graphical interface. Experience with Python is therefore expected when using *datafold*.

3.2.3 Measures for sustainable software development

Developing durable scientific software that provides machine learning to solve general purpose problems is a challenging task [Anzt et al., 2020; Sculley et al., 2015]. Under time pressure, especially with the urgency to publish results, it is always tempting to write sub-optimal code that "just works" to proceed with a task. While this is sufficient for source code that only serves a single purpose, a sub-optimal increment degrades the overall quality of software that aims to be reusable and extensible. An established metaphor in software engineering is *technical debt*: *the term refers to future obligations that are the consequence of technical choices made for a short-term benefit*. [Hinsen, 2015, p. 1]. Analogous to financial debt, technical debt is not always bad, but it is important to know that it is present in a project and how severe it is. Continuous software maintenance strives to not let debt accumulate to an unmanageable degree [Goble, 2014; Sculley et al., 2015]. In particular, software that is actively developed with many ideas of future directions and algorithmic experimentation can incur high ongoing maintenance costs. Servicing debt involves refactoring code, improving unit tests, removing old code, reducing software dependencies or improving documentation [Fowler, 2018; Sculley et al., 2015].

For *datafold* I strive for a *sustainable* software in the sense of Anzt et al. [2020]. To keep development time and technical debt to a practical minimum, I make use of additional tools and procedures [Renggli et al., 2019]. All source code and associate files are hosted on a publicly accessible repository on "gitlab" [1]. While gitlab is a platform to host source code, it also comes with many additional features that support software management.

Continuous Integration (CI) pipelines constitute an integrative and indispensable tool of modern software engineering [Renggli et al., 2019]. A CI pipeline systematically manages the development efforts and performs various automatable tasks to verify and check that the software meets certain criteria. Every contributed change of a developer is tagged as pass, warning or fail. Only code changes that pass are accepted to be integrated into the main code basis. Fig. 3.3 gives an overview of *datafold's* CI pipeline. The entire pipeline is specified in the file `.gitlab-ci.yml` as part of the source code itself.

[1] gitlab.com/datafold-dev/datafold

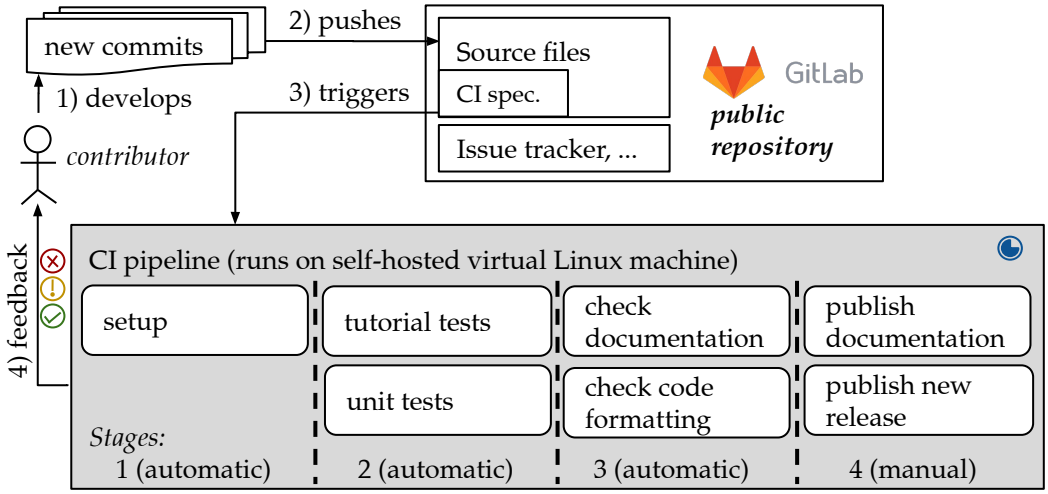


Figure 3.3: Schematic overview of the CI pipeline in *datafold*. The pipeline includes tests and checks that foster an organized and sustainable future development of *datafold*.

The following list includes measures to promote sustainable software management for *datafold*, which also includes the steps of the CI pipeline. The numbers in brackets correspond to points for sustainable software according to Anzt et al. [2020].

Citable and licensed software (2, 9)

Datafold is published in the "Journal of Open Source Software" (JOSS) [Lehmberg et al., 2020a]. During the publication process of JOSS both an article and the software itself are reviewed by peers [Smith et al., 2018]. As such, the package is citable in a traditional way of a scientific contribution. The source code of *datafold* is distributed under the permissive and academic "MIT license"[2]. With these terms, the software can be used or manipulated as a whole or in parts by anyone who wants to contribute to the original project or continue development in a separate repository.

Version control and tagged releases (7, 15)

Versioned software makes it easy to recover past snapshots of the source code to document the development process and reproduce results. All changes to files in the repository are tracked with the widely-adopted source control management "git"[3]. Additionally, *datafold* collects multiple code changes in tagged releases in a semantic version scheme, *major.minor.patch* (e.g. 1.1.6). An increase in the major number indicates breaking changes in the API, an increment in the minor number of new features and increasing the last number highlights bug fixes and small improvements. The version that I describe and use in my thesis is 1.1.6.

Document bugs and open tasks in issue tracker (7)

[2]https://choosealicense.com/licenses/mit/
[3]https://www.git-scm.com/

A danger of technical debt is that it accumulates without the software maintainers knowing where it lies. Issue trackers are established in many open-source projects to document open tasks. An issue describes an isolated task, in which the progress of the task is recorded. This can include necessary refactoring, bug fixes and new features. For *datafold* the issue tracker is managed in "gitlab" and is available at gitlab.com/datafold-dev/datafold/-/issues/.

Publicly available software (6)

Making scientific software code publicly available beyond a research group has many advantages, such as avoiding repetitive software development in a community or increased trust by publicly reproducing results [Anzt et al., 2020]. In addition to the "gitlab" repository where the software development is recorded, each *datafold* version is also available at the official "Python Package Index" (PyPI)[4]. The installation of *datafold* is then straightforward with Python's packaging tool `pip` in the command line:

```
pip install datafold
```

A new release of *datafold* is uploaded to PyPI if the steps in stage 4 of the CI pipeline are manually executed (Fig. 3.3). This is only possible when all previous stages in the CI pipeline pass.

Interoperable API, avoiding large re-implementations of functionality and dependency management (10, 13, 16)

As highlighted in *datafold's* software architecture, the implemented methods use the existing functionality of Python's scientific computing stack and align their API for data-driven methods to the widely-adopted *scikit-learn*. The packages are not part of the standard library and need to be installed separately. All dependencies are managed in a standardized file `requirements.txt`, which is processed for automatic dependency management. Further dependencies for the software development are contained in the file `requirements-dev.txt`.

Separate productive and experimental code

As a scientific software *datafold* should also be usable and encouraging to perform algorithmic experimentation because it is often not clear beforehand which algorithm is best suited. With this measure, I try to find a pragmatic way to balance the anti-pattern "dead experimental code" [Sculley et al., 2015] and the reality of software development in research [Wiese et al., 2020]. I organized experimental and production code within *datafold*, whereby I explicitly flag classes that have not reached high standards. These classes are not part of the documentation and raise a warning when used.

Documentation and tutorials (8, 11, 19, 21)

[4] https://pypi.org/project/datafold/

The documentation of a software includes the description of the API, tutorials, literature references, and instructions for new developers. The entire documentation is hosted on a web page[5], which is updated with every new versioned release of *datafold* (see stage 4 of the CI pipeline, Fig. 3.3). The HTML pages are generated with the Python package *Sphinx*[6].

Unit tests and coverage report (14)

All productive code in *datafold* requires unit tests, which are managed along with the source code in separate directories. All tests can be executed with the Python packages *pytest*[7] and *coverage.py*[8]. The former executes the test and the latter provides an additional report in a web page format to highlight which lines in the source code are part of a test[9]. As of *datafold* version 1.1.6 there are 340 tests with a coverage rate of 88%. All tests and tutorials run in the CI stage 2 in Fig. 3.3.

Source code formatting

All source code in *datafold* is automatically formatted in a deterministic and consistent way with the two tools *black*[10] and *isort*[11]. This increases tremendously the code readability and is a way to reduce technical debt. Stage 3 of the CI pipeline verifies whether the source code is correctly formatted according to the rules implemented in the two tools.

3.3 *pcfold*: Data structures for point cloud manifolds

In this section, I start with *pcfold* as the first layer of *datafold's* software architecture. Fig. 3.4 outlines the main components in *pcfold*. The layer includes a new data structure `TSCDataFrame` to capture time series collections, as observations from dynamical systems. The data structure is a central element for many methodological generalizations in the higher layers. Moreover, there is a diverse set of basic functionalities that directly operate on `TSCDataFrame`. This includes splitting time series in training and validation data or computing a metric between a true and predicted time series. Both of these functionalities are necessary for model validation and parameter optimization within the system identification loop (cf. Fig. 2.9).

[5] https://datafold-dev.gitlab.io/datafold/
[6] https://sphinx-doc.org/
[7] https://docs.pytest.org/
[8] https://coverage.readthedocs.io/
[9] latest report: https://gitlab.com/datafold-dev/datafold/-/jobs/artifacts/master/file/coverage/index.html?job=unittests
[10] https://black.readthedocs.io/
[11] https://pycqa.github.io/isort/

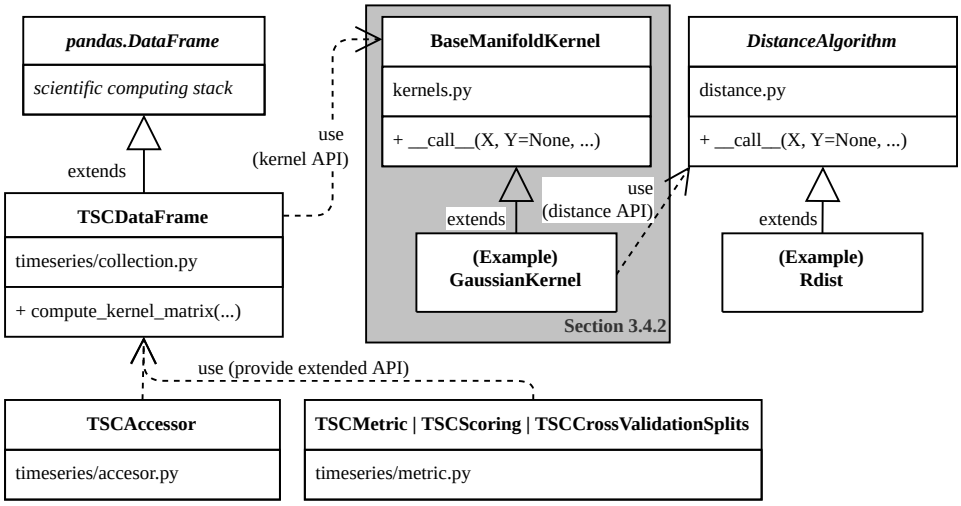


Figure 3.4: Class diagram of `TSCDataFrame` as a data structure to store time series collections. Around the data structure are additional functionality required for kernel-based methods and model validation. The grayed area is covered in Section 3.4.3 within Diffusion Maps (DMAP) as a kernel-based method.

An important functionality in *pcfold* is to provide an algorithmic framework to compute a kernel matrix from data (with or without temporal context). This kernel interface together with the data structure serve as building blocks for kernel-based methods such as the DMAP included in the *dynfold* layer.

As outlined in Eq. 2.35 – 2.36, setting up a kernel matrix requires (1) computing a distance matrix (dense or sparse) and (2) applying a kernel function. Both the optimality of a metric and kernel depend on the underlying system and dataset. Therefore, there are many algorithmic variations described in the literature that change either of the components [e.g. Berry and Sauer, 2019; Giannakis, 2015]. For this reason, the *pcfold* layer includes base classes that provide a flexible and easy integration of specialized algorithms that also go beyond the basic functionality of the scientific computing stack. For improved readability of the thesis, I postpone describing the kernel interface to Section 3.4.3 where it is required within the DMAP method. Section 3.3.2 covers the interface to compute distance matrices, which also includes a new algorithm to efficiently compute a sparse δ-range distance matrix from data with a manifold assumption. I implement this algorithm in a separate software *rdist* as an additional contribution of my thesis.

3.3.1 Data structure for time series collection

In contrast to static data, where samples can be treated independently, the temporal context in time series data requires additional meta-information — the time context — for each sample. This introduces new dataset characteristics, such as the sampling frequency (arbitrary or equidistant), number of time series (a single coherent or many separate) and number of features (univariate or multivariate).

While the *NumPy* array is the standard data structure for static data, there is no such widely-accepted data structure for processing time series. One challenge is that the additional time information introduces new variations of how the data is organized, as it is inherently ordered. I suspect that many data structures for time series formats are created with the assumption of the respective algorithm in mind that operates on the data structure. As a result, it is hard to re-use the data structure for a different algorithm with diverging assumptions. Moreover, I found that many proposed data structures for time series also become incompatible with the standard alignment of static data in computer memory. This is a disadvantage because additional copy operations are needed to translate between the two data types.

In this section, I propose my own data structure `TSCDataFrame` to store time series *collections*. In Section 3.3.1.1 I show that I could overcome many limitations that I found in alternative data structures. The new specification was necessary to generalize the methods for system identification in higher layers of *datafold*. Particularly algorithms associating to the DMD are usually able to handle multiple time series [Tu et al., 2014; Williams et al., 2015]. In the Sections 3.3.1.2 and 3.3.1.3 I highlight the management within *pcfold* of additional basic functionalities that directly relate to the data structure and are needed for data-driven modeling.

3.3.1.1 TSCDataFrame

The data structure's name `TSCDataFrame` is a composite of the abbreviation TSC (time series collection) and the base class `DataFrame` from the Python package *pandas* [McKinney, 2011]. An essential feature of `DataFrame` is that it stores data in a tabular form, which allows meta-information to be attached to each point sample. For time series data this is mainly required to store time information. While *pandas* already provides functionality for time series, its main focus lies on handling date-based indices. The additional value of `TSCDataFrame` is in providing necessary functionality and data organization for data-driven modeling and system identification.

Fig. 3.5 displays the main data-driven scenarios in which time series collections become relevant. In the left image **(a)** a time series is observed over time where all states have a *unique* time value. If the measurement process is interrupted and re-started, a new time series is sampled after an interval of missing data. This is a typical scenario for real-world sensor measurements, where each sample has an associated timestamp. In the second case **(b)** of Fig. 3.5 all initial states share the same normalized time reference (e.g. $t_1 = 0$). The time values in the series are no longer unique in the global reference of the time series collection. In contrast to the first case, this occurs if a dynamical system is systematically sampled with initial states that all associate with the same initial time. A typical scenario is when a simulation software is sampled to generate a surrogate model [e.g. Dietrich et al., 2018]. If the dynamical system is autonomous, case (a) can be transferred to case (b), however, this loses the global order across the entire collection. The third and last case **(c)** is a combination of the previous cases. This

sampling scenario occurs, for example, if a time interval is removed from time series collection to perform cross-validation [Bergmeir and Benítez, 2012].

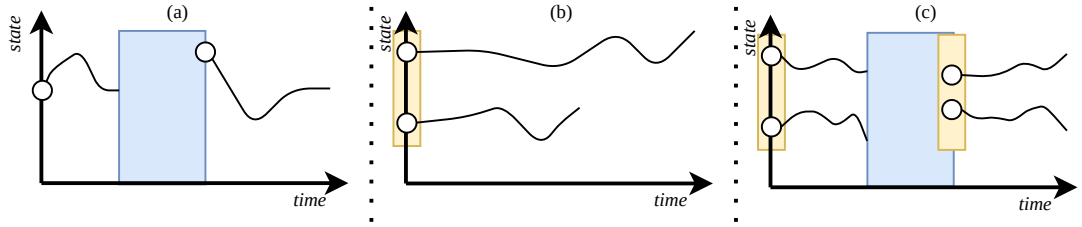


Figure 3.5: The main scenarios of time series collections to use `TSCDataFrame`. All (multivariate) time series have an initial condition (circle) and share the same spatial features. The time series in a collection can be of different lengths. A blue block between time series corresponds to a time interval of missing data, while a yellow block of initial states denotes an equal time reference.

Table 3.1 gives a concrete example of the layout of `TSCDataFrame`. The time series collection is from the pendulum system covered in Section 4.1. Because all initial states have a time value $t_1 = 0$, the collection corresponds to the middle case (b) in Fig. 3.5.

The right side of Table 3.1 showcases additional basic functionality, which is attached to `TSCDataFrame` to better describe the dataset. For example, the first statement queries the number of time series contained in the collection and the attribute `delta_time` describes the equidistant time sampling rate across the time series. The last statement (`to_numpy()`) removes all row and column indices and returns a *NumPy* array with the standard alignment of static data. The function is already provided by the base class `DataFrame` and returns a view of the internal data without copying it (if all columns have the same data type). A conversion from `TSCDataFrame` to the standard format is therefore possible with no additional cost.

Table 3.1: Example of a `TSCDataFrame`.

ID	feature time	x	y
0	0.000000	0.87	0.50
	0.050366	0.74	0.33
	0.100732	0.58	0.18
	...	...	...
1	0.000000	-0.71	1.71
	0.050366	-0.78	1.63
	0.100732	-0.85	1.52
	...	...	...
2	0.000000	1.00	1.00
	0.050366	0.99	0.89
	0.100732	0.97	0.76
	...	...	...

```
tscdf: TSCDataFrame
tscdf.n_timeseries
>>> 3
tscdf.n_features
>>> 2
tscdf.delta_time
>>> 0.050366
tscdf.n_timesteps
>>> 500
tscdf.is_equal_length()
>>> True
tscdf.is_const_delta_time()
>>> True
tscdf.is_same_time_values()
>>> True
static_data = tscdf.to_numpy()
>>> np.array([[0.87 0.5 ],
              [0.74 0.33],
              ...         ])
```

As highlighted before, the main idea of `TSCDataFrame` is to inherit from `DataFrame` as a rich data structure. In `TSCDataFrame`, the generic form is restricted to the special layout of a time series collection which has to fulfill certain criteria. The major advantage of class inheritance over composition is that `TSCDataFrame` remains fully compatible with *pandas* and, therefore, profits from its sophisticated hierarchical indexing capabilities. The time series collection can easily be processed in a per state, time value, feature or time series fashion [McKinney, 2011]. The special format of `TSCDataFrame` has the following criteria (see Table 3.1 as a concrete example):

1. Each row in `TSCDataFrame` corresponds to a single state measurement. All values must be numeric (integer, real or complex), which also includes invalid numbers "not a number" (nan) or infinity (inf). Other common data types such as strings or general objects are disallowed in `TSCDataFrame`.

2. The row index must be a multi-index of two levels. The first level indicates the time series ID with a positive integer and the second level contains the time values. Each time series within the collection must be coherent. The time values in each time series must be non-negative and unique numbers sorted in ascending order.

3. The column index must only contain a single level to identify the spatial features (numeric or string). In Table 3.1 this is the **x** and **y** to indicate the respective column. All index names must be unique.

In a mathematical notation, `TSCDataFrame` captures time series collection data in the form of

$$X = \left[\mathbf{x}_1^{(1)}, \ldots, \mathbf{x}_{J_1}^{(1)} | \ldots | \mathbf{x}_1^{(I)}, \ldots, \mathbf{x}_{J_I}^{(I)}\right] = \left[X^{(1)}, \ldots, X^{(I)}\right], \tag{3.1}$$

where $\mathbf{x}_j^{(i)} = [x_1, \ldots, x_N]_j^{(i)}$ is an N-dimensional column state (or snapshot) with associate time indices i (ID) and j (time index); see Section 2.3.2 for the specification in the EDMD framework. However, the representation of `TSCDataFrame` and Eq. 3.1 differ for their respective context: storing and processing data on a computer versus readability of mathematical notation. In `TSCDataFrame` the data alignment is row-wise, while it is column-wise in Eq. 3.1. The orientation in the data structure follows the conventional data alignment of the scientific computing stack. Note that *scikit-learn* also supports `DataFrame` as input data and therefore also `TSCDataFrame` as a subclass[12].

The rows in `TSCDataFrame` are required to be indexed with two layers to actually index a time series *collection*. For each row the tuple (i, t) is stored to be able to capture all three cases of Fig. 3.5. On the other hand, the mathematical notation in Eq. 3.1 typically works with an index of the time series (i) and time index (j). Both elements map to the respective timestamp $(i, j) \to t$.

Overall, `TSCDataFrame` supports any number of ...

	parameter in Eq. 3.1
... features	$N \geq 1$
... time series	$I \geq 1$
in the collection, as well as ...	
... arbitrary time values	$t_j^{(i)} \geq 0$
... and number of time steps	$J_i \geq 1$
in each time series of the collection.	

Note that for practical reasons a time series can also consist of a single sample, i.e. $J_i = 1$. Although a valid time series requires at least two samples, this is permitted to use the data structure in situations where only a single state per time series is required. For example, this is the case to describe the initial condition state of each time series.

The above features of `TSCDataFrame` and the strong integration to *pandas* — and ultimately compatibility to many packages of the Python scientific computing stack — overcome limitations that I found in other Python software projects. Table 3.2 provides a comparison of multivariate time series *collections*. These projects already generalize the two main limitations in many other projects: (1) only support univariate time series and (2) only support single coherent time series.

[12] Currently, *scikit-learn* only supports `DataFrame in, NumPy array out`, i.e. without forwarding the indices to the output. However, the maintainers intend to extend the API for `DataFrame in, DataFrame out`. This behavior was anticipated in *datafold* and if the handling becomes available in the future, the `TSCDataFrame` integrates seamlessly.

Table 3.2: Comparison of proposed data structures that model time series collections. (arbit. = arbitrary, contig. = contiguous, comp. = compatibility)

Package	Object	Varying length	Arbit. time	Contig. memory	NumPy comp.	sklearn comp.
tslearn[1]	3-dim. `ndarray`	no	no	no	yes	no
pysindy[2]	list of 2-dim. `ndarray`	yes	no	no	no	no
darts[3]	composition of `DataFrame`	yes	yes	yes	no	no
sktime[4]	nested `DataFrame`	yes	yes	no	no	no
deeptime[5]	composition of `ndarray`	yes	no	no	no	no
datafold `TSCDataFrame`	multi-indexed `DataFrame`	yes	yes	yes	yes	yes

[1]Tavenard et al. [2020]; [2]de Silva et al. [2020]; [3]Herzen et al. [2021]; [4]Löning and Király [2020]; [5]Hoffmann et al. [2021]

The two main approaches in the alternative data structures of Table 3.2 to capture time series collections are to (1) generalize the *NumPy*, `ndarray` or (2) make use of `DataFrame` from *pandas* (which is also my approach). The first two columns in Table 3.2 compare the objects to indicate whether a respective data structure allows time series of variable length and arbitrary sampling. For example, *tslearn* describes a time series collection in a three-dimensional tensor, which forces all time series to have the same length (or fill invalid numbers). The next column of Table 3.2 indicates whether the data is stored contiguously in memory. If this is not the case, the object requires inefficient copy operations to bring the data in a correct form, before it can be processed with efficient numerical algorithms as in *NumPy* [Harris et al., 2020]. The last two columns include example operations from *NumPy* and *scikit-learn* to highlight whether a data structure is compatible with the respective package,

```
# X: respective data structures in Table 3.2
np.linalg.svd(X)
sklearn.MinMaxScaler().fit_transform(X)
```

Both operations are applied directly on the respective data structure `X` as an indication of the compatibility —- I only mark whether the operation raises an error (no) or passes (yes).

Of course, `TSCDataFrame` is yet another data structure. However, Table 3.2 highlights how my proposed data format overcomes many common limitations within alternative formats. Furthermore, the data structure is essential to efficiently address some of the features of DMD-based methods. But also from a software engineering

perspective, the data structure is beneficial because it encapsulates various forms of sampling strategies while storing the data efficiently and remaining compatible with existing projects.

The next section describes the organization of extended functionality that is required within method implementations that make use of `TSCDataFrame`. Section 3.3.1.3 then describes basic functionality required within the data-driven modeling workflow.

3.3.1.2 Extended functionality: `TSCAccessor`

The *pandas* package provides various ways to extend its main data structures[13]. Inheritance is one way that I use for `TSCDataFrame`. Another way is to organize domain-specific functionality that associates to the data structure in so-called "accessors". An accessor is registered and attached to the data structure to provide additional functionality.

In *pcfold,* I include a `TSCAccessor` that attaches specific functionality for the data structure `TSCDataFrame` in the additional attribute `tsc`. This allows me to clearly separate and organize dedicated methods that are often needed when using the data structure. The following code snippet sketches some methods in the `TSCAccessor` and at the end gives an example of its usage:

```
@pd.api.extensions.register_dataframe_accessor("tsc")
class TSCAccessor():

    def __init__(tsc_df: TSCDataFrame):
        self._tsc_df = tsc_df

    def check_tsc(...)
            # Ex. 1: check_tsc(ensure_const_delta_time=True)
            # Ex. 2: check_tsc(ensure_min_samples=10)

    def normalize_time() [...]
    def time_derivative() [...]
    def shift_matrices() [...]

# Example: first access attribute "tsc" and then a method
X: TSCDataFrame
X_minus, X_plus = X.tsc.shift_matrices()
```

The first function `check_tsc(...)` provides an easy way to validate specific assumptions about input data stored in `TSCDataFrame`. For example, whether all time series are equidistantly sampled (Ex. 1) or if all time series have a minimum number of samples (Ex. 2). The next function `normalize_time(...)` returns a `TSCDataFrame`

[13] https://pandas.pydata.org/pandas-docs/stable/development/extending.html?

with normalized time, such that all time series in the collection have time values oriented to a (global) reference time starting at $t = 0$ and a sampling rate of $\Delta t = 1$. With `time_derivative(...)` it is possible to approximate time derivatives with finite difference schemes of arbitrary order. Setting up the scheme and computing the derivative is performed with the Python package *findiff* [Baer, 2021]. The last function `shift_matrices(...)` performs a typical operation that is required for DMD-based methods. It creates the shifted matrices required for a regression in a linear system identification (Eq. 2.28 on page 34). At the end of the code snippet, the shifted matrices are computed by compiling the shift matrices from all time series in a `TSCDataFrame`.

Overall, the accessor provides a straightforward approach to collect specific functionality for time series collection data, that is not intended to be attached to the actual data structure.

3.3.1.3 Extended functionality: validation splits and error metric

In data-driven modeling it is important to split the available data into separate parts to obtain an unbiased model validation. Typically, the available data to construct a model is split into *training* data and optimized on separate data, referred to as the *validation* data. The training error (i.e. how well the model reconstructs the data) and validation error (i.e. how well the model predicts out-of-sample scenarios) provide valuable insight into how well the model approximates the underlying system. For example, the discrepancy between the training and validation error indicates how well the bias-variance trade-off is balanced [Murphy, 2012, Sec. 6.4.4]. A large discrepancy indicates that the model overfits the data.

While there are many established procedures for static data, these do not directly transfer to the case of time series collection data. Importantly, the data is no longer independent and cannot be treated as such. For example, data splits need to maintain characteristics of the time series data, such as a constant time sampling. Fig. 3.6 (left) exemplifies different ways of how a time series collection can be split.

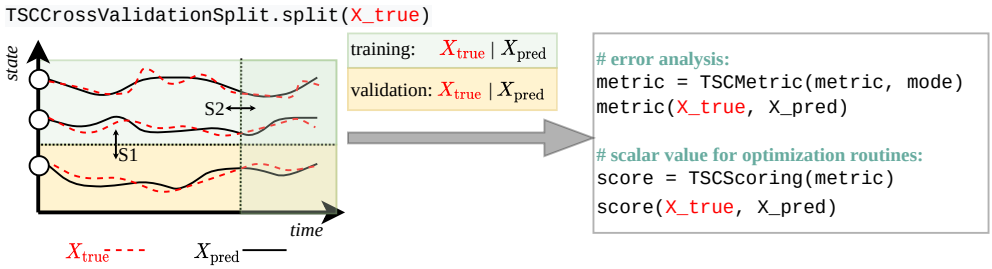


Figure 3.6: Overview of basic functionality required for model validation on `TSCDataFrame` between true and predicted time series. With `TSCCrossValidationSplit` the data can be split either along the time series in a collection (S1) or along the time axis (S2). Based on the split, the differences between true and modeled time series can be evaluated with the additional classes `TSCMetric` and `TSCScore`.

In *pcfold* three dedicated classes perform the basic operations for model validation: (1) split the data, (2) compute a metric and (3) score a model. The new class definitions within *datafold* are necessary to account for the special handling of time series collection data. However, all three classes mimic the behavior of *scikit-learn*[14] to remain compatible with the established data-driven workflow. The following list briefly describes all three operations:

- `TSCCrossValidation` is a base class to split data stored in a `TSCDataFrame`. Usually, this is performed on the measurements X_{true} to obtain separate sets for training and validation. Unlike for static data, it is important to consider the temporal context in the splits and maintain the necessary properties for a method, e.g. a constant sampling interval Δt. Fig. 3.6 includes two typical splits. The first split S_1 is performed along the initial conditions and implemented in `TSCKFoldSeries`. The second split S_2 is provided by `TSCKFoldTime` and splits along the time axis. Which splitting strategy is suitable in a concrete setting depends on the underlying system properties (e.g. transient or ergodic state evolution) and the sampling form as per Fig. 3.5. For example, splits along the time axis are more suitable for long time series in case (a), whereas splits along time series are more suited for case (b).

- The class `TSCMetric` computes a metric between two `TSCDataFrame` objects, corresponding to true measurements X_{true} and predicted data X_{pred}. Common metrics for time series analysis are the root mean squared error (RMSE) or the mean absolute error (MAE). Contrasting to static data, however, the time information introduces more "dimensions" in how a metric can be evaluated: per time series, per spatial feature or per time step. These cases are covered in the argument `mode` of the metric. The following list exemplifies the handing for the RMSE metric:

```
# compute metric for each time series
# requires: multiple time series (I > 1)
# returns: an aggregated error for each time series
rmse = TSCMetric(metric="rmse", mode="timeseries")
rmse(X_true, X_pred)

# compute metric for each time step
# requires: multiple time series with same reference time
#           (case (b) in Fig. 3.5)
# returns: an aggregated for each time step
rmse = TSCMetric(metric="rmse", mode="timestep")

# compute metric for each feature column
# requires: multivariate time series (N > 1)
# returns: an aggregated error for each feature (column)
rmse = TSCMetric(metric="rmse", mode="feature")
```

[14] https://scikit-learn.org/stable/modules/cross_validation.html

To compute a metric both objects in the argument must have identical indices. Note that once a metric object is initialized, it is *callable*. For example, the variables in the code snippet can be used with `error = rmse(X_true, X_pred)`.

All three modes can highlight different characteristics of the discrepancy between a model and observation data. All modes are therefore suitable for an extended error analysis within the model validation (cf. the system identification loop in Fig. 2.9). For example, if the metric is computed per time step this again results in an `TSCDataFrame` object, which can be used to identify systematic time-dependent patterns of the error, such as periodic changes or steady increases of the error. Such error analysis can also be used to detect regime transitions in a system as highlighted in Gottwald and Gugole [2020].

- `TSCScoring` performs a final aggregation of a specified metric to a *scalar* value. The computed metric is averaging to a single value. As per the metric, a score is a callable object and requires two `TSCDataFrame` objects with identical indices. According to *scikit-learn*, a score is interpreted as "higher is better". This means error metrics such as RMSE are negated. A score is suitable as the objective to maximize within parameter optimization routines as part of the model selection.

3.3.2 Computing dense or sparse distance matrices (with Rdist)

This section continues with an interface in *pcfold* to compute a distance matrix from data. As highlighted in Section 2.4.1 — in the context of kernel eigenmaps — a distance matrix is the first step to set up a neighborhood graph to describe a discrete version of the underlying manifold. The structure of a distance matrix therefore affects the extracted geometry. Furthermore, the algorithm to compute the distance matrix largely influences the computational cost. A dense distance matrix scales $\mathcal{O}(J^2)$ (where J are the number of points) both in computational time and computer memory. With a typical laptop memory of 16 gigabytes (GB), this permits a maximum number of samples of about $J \approx 44000$. The typical stages of escalation to better scale kernel-based methods are:

1. **small to medium** — computing and storing all distance pairs in a dense matrix is possible.
2. **medium to large** — only relevant nearest neighbors (k-NN or δ-range) are computed and stored. (What constitutes a *relevant* neighbor is will depend on the kernel choice.)
3. **large** — subsample the original point cloud by selecting reference points (also referred to as landmarks) [Shen and Wu, 2020]. A danger is that sparsely sampled regions in a point cloud vanish in the reference points.

Sparse matrices provide a first way to improve the memory complexity to $\mathcal{O}(k \cdot J)$, where $k \ll J$ is the (average) number of neighboring pairs that are stored per point.

Moreover, there are efficient algorithms dedicated to space matrices in follow-up operations, such as computing the eigenpairs from a sparse kernel. In my thesis, I only consider cases 1 and 2.

In the next subsection (3.3.2.1) I describe the general interface in ***pcfold*** that provides an easy integration of alternative implementations of either dense or sparse distance matrix computations. One such available implementation is ***rdist*** as a contribution of my thesis. The underlying algorithm of ***rdist*** to compute a sparse δ-range distance matrix is described in Subsection 3.3.2.2.

3.3.2.1 General distance matrix interface

The ***pcfold*** layer provides an interface that allows a dense or sparse distance matrix to be computed from a point cloud (which can also be a `TSCDataFrame`). In the dense case, a ***NumPy*** array is returned, whereas in the sparse case a matrix in a compressed sparse row (CSR) format is returned; as provided by the ***SciPy*** package [Virtanen et al., 2020].

While computing a dense distance matrix is straightforward in a brute force fashion, there are many different approaches and implementations to compute a sparse distance matrix. For the sparse case I only consider δ-range algorithms (see Section 2.4.1). The main advantage is that a pairwise kernel matrix is symmetric for both the dense or δ-range case (cf. Table 2.4). However, the interface in ***pcfold*** is easy to extend to also support k-NN algorithms.

Fig. 3.7 displays algorithms available in ***pcfold***, which all derive from a single abstract base class `DistanceAlgorithm`. The interface consists only of a single function `__call__(X,Y)`, which in Python is a so-called magic function. In this case, the function makes an object directly callable without the need to access an explicit method name [VanRossum and Drake, 2010].

A general distance matrix is computed from two static point clouds: (1) reference points (X) and (2) query points (Y) (see Curtin et al. [2013]). A component-wise distance matrix contains all distance values from points included in Y to points in X. The special case in which the reference and query points are identical $Y = X$ is referred to as the pairwise case:

$$D_{i,j}(X, X) = d(\mathbf{x}_i, \mathbf{x}_j) \qquad \text{pairwise (training)} \tag{3.2}$$

$$D_{i,j}(X, Y) = d(\mathbf{x}_i, \mathbf{y}_j) \qquad \text{component-wise (out-of-sample, test).} \tag{3.3}$$

In kernel-based methods, the reference points X are always the training data, whereas Y contains out-of-sample data. Typically, the two cases of a pairwise and component-wise distance matrix are treated separately. This is because the pairwise case allows for algorithmic optimizations as the final matrix is symmetric and square. For example, to compute a dense pairwise distance matrix in the ***SciPy*** package the function `pdist(X)` is applicable, whereas for the component-wise case the `cdist(X,Y)` is used.

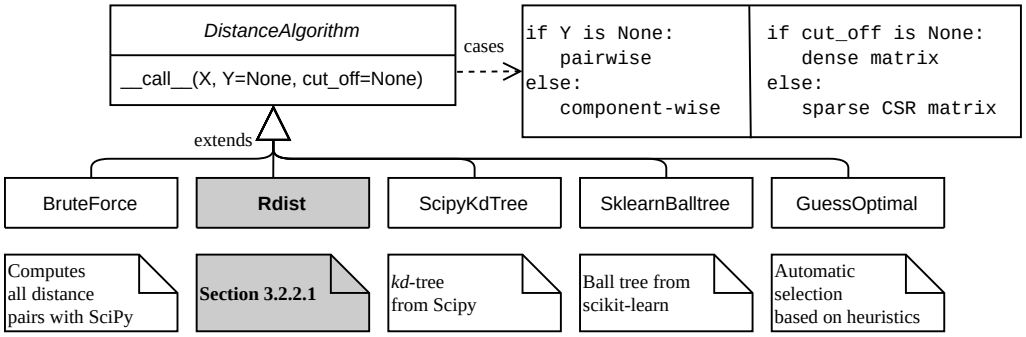


Figure 3.7: Class diagram for distance algorithms in *pcfold*. The `DistanceAlgorithm` provides a base class which requires a implementation of the `__call__` function in the subclasses. The algorithm highlighted in gray is described in Section 3.3.2.2 as part of my thesis contribution.

The default setting in *pcfold* is to choose an algorithm based on a fixed set of rules in `GuessOptimal`. Here an algorithm is selected based on simple heuristics that consider the dataset size and the parameter choices. If no `cut_off` is set (`cut_off` corresponds to δ) then a dense matrix is computed, otherwise a sparse distance algorithm is selected. An optimal selection of δ depends on the relative distances and properties of the dataset. Moreover, its choice is also constrained by computer memory for large datasets.

The algorithms to compute a sparse distance matrix usually store the point cloud in tree-based data structures [e.g. Muja and Lowe, 2014]. These data structures already provide a coarse neighborhood relation within the point cloud and can therefore limit the necessary number of distance computations. In Fig. 3.7 a kd-tree and ball tree implementation from *SciPy* and *scikit-learn* are included. While such standard approaches are a good choice for relatively low point dimensions, the search time often grows substantially for increasingly larger point dimensions (i.e. $N \gg 1$). The next section describes a new algorithm as a way to speed up existing δ-range algorithms for this case of larger point dimensions.

3.3.2.2 Rdist: Efficient computation of sparse distance matrix for high-dimensional data

This section describes a new algorithm that I developed together with Felix Dietrich. Readers who are primarily interested in *datafold* and its architecture may skip this section.

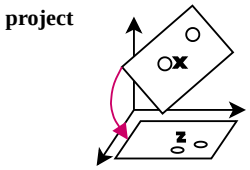
Rdist is an algorithm to efficiently compute a sparse δ-range distance matrix with Euclidean metric from high-dimensional data. The idea is a result of algorithmic experimentation to speed up existing distance matrix algorithms and ultimately kernel-based methods, such as Diffusion Maps covered in the next section. Unlike probabilistic approaches such as Locality Sensitive Hashing (LSH) [Andoni and Indyk, 2006] or ap-

proximate nearest neighbor algorithms [Muja and Lowe, 2014], *rdist* is exact in that it can be proven to find *all* neighbors within a δ radius for each point.

The main idea for speeding up the neighborhood search is based on a simple observation: linear and unitary projections of a point cloud always preserve the local neighborhood. This is given by the Johnson-Lindenstrauss lemma for random projections [Clarkson, 2008]. The manifold assumption is central to the idea, which states that high-dimensional points within a dataset are assumed to be sampled on a manifold with intrinsic lower dimension (cf. Section 2.4.2). Algorithm 1 describes *rdist* with graphics to illustrate the three essential steps.

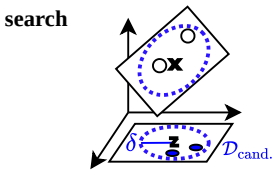
Input : $X \in \mathbb{R}^{[J \times N]}$ point cloud with points $\mathbf{x} \in \mathbb{R}^N$ oriented row-wise
Parameter: $\delta > 0$, radius
$\gamma \in (0, 1]$, contraction bound
$S < J$, number of samples to draw in PCA
Output : $\mathcal{D}$, sparse distance matrix in compressed sparse row (CSR) format

1. $Z = \texttt{PCA(n\_components=R).fit\_transform}(X_S)$
where $R = \sum_{i=1}^{k} \sigma_i \geq \gamma \bar{\sigma}$ with smallest integer k that fulfills the condition; the eigenvalues are sorted ($\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_N$) and $\bar{\sigma} = \sum_{i=1}^{N} \sigma_i$



scikit-learn
(Python package)

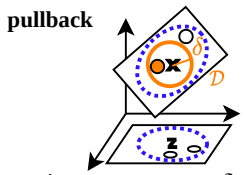
2. $\mathcal{D}_{\text{cand.}} = \{(i, j) : ||\mathbf{z}_i - \mathbf{z}_j|| < \delta, (\mathbf{z}_i, \mathbf{z}_j) \in Z\}$



fast δ-range search algorithm
(shared library in C++)

mlpack

3. $\mathcal{D} = \{(i, j, d) : (i, j) \in \mathcal{D}_{\text{cand.}}, d = \|\mathbf{x}_i - \mathbf{x}_j\|_2 < \delta\}$



own implementation
(translated and compiled C code)

Algorithm 1: *Rdist*: Computing a sparse δ-range distance matrix with manifold assumption. Note that the PCA is expressed in terms of *scikit-learn* code notation for easier readability.

Since the distance matrix is computed independently of temporal order in the data, the time indices are omitted here. The data corresponds to $X \in \mathbb{R}^{[J \times N]}$ with J samples of

N-dimensional states.

The idea is expressed in three steps and referred to as "project-search-pullback":

1. **Project** the high-dimensional point cloud into a low-dimensional space with a random and unitary matrix. In the illustration for step 1, this is exemplified with a projection from a two-dimensional plane in three dimensions ($\mathbf{x} \in X$, corresponding to high-dimensional samples on the manifold) into a reduced space of two dimensions ($\mathbf{z} \in Z$).

   To increase the likelihood that the point distances are well-preserved during this projection, I perform a Principal Component Analysis (PCA) based on a subsampled dataset X_S (containing S randomly selected points). While it is possible to use *any* random unitary matrix, using the (approximate) principal coordinates is justified by linear dimension reduction, where the principal components orientate to directions with maximal variance in the dataset [Murphy, 2012, Sec. 12.2]. Another benefit of computing the PCA is that it also computes eigenvalues $\{\sigma_i\}_{i=1}^{N}$, which can be used to compute a target dimension R of the projection (in the graphic $R = 2$). For this, I include the contraction bound, $\gamma \in [0, 1]$, as a new parameter that steers the trade-off between computational cost and the preservation of distance values according to the PCA eigenvalues.

2. **Search** and collect all point pairs (i, j) within the δ-radius in the projected space (Z) and store the candidate pairs in $\mathcal{D}_{\text{cand.}}$. Note that $\mathcal{D}_{\text{cand.}}$ does not store the actual distances in the projected space. Because the point dimension is reduced, the search can now be performed more efficiently by standard algorithms such as the kd-tree.

   Crucially, according to Johnson-Lindenstrauss lemma, all pairwise Euclidean distances in the projected space are *equal or smaller* to the (original) ambient space distances. From this follows that the set of candidate pairs in $\mathcal{D}_{\text{cand.}}$ is *guaranteed* to contain all true pairs of the ambient space. In the illustration of Algorithm 1 the δ-radius contains two points, which correspond to an ellipse (in blue) in the ambient space.

3. **Pullback** the candidate point pairs $\mathcal{D}_{\text{cand.}}$ to the ambient space. Essentially, this step corresponds to a filter operation in which all candidate pairs are removed that have a distance greater than the radius $d_{i,j} > \delta$ in the ambient space. Ideally, the number of pairs to be removed is small, which depends on how well the distances are preserved during the projection of the first step. If many points are removed then this introduces a larger computational overhead. Finally, once the candidate pairs are filtered, the distance matrix $\mathcal{D}$ is set up in a sparse format and returned by the algorithm.

The basic form of the algorithm is relatively easy to implement because the search is executed by existing δ-range algorithms. *Rdist* is therefore not an algorithm to search nearest neighbors *per se*, but provides an approach to speed up computations of existing

algorithms if the manifold assumption is fulfilled. If a more efficient δ-range algorithm becomes available, it is possible to incorporate it in *rdist*.

To justify the additional operations in "project" and "pullback", the main objective in *rdist* is enhancing computational speed. However, as highlighted in Section 2.5.2, Python itself has drawbacks for efficient low-level algorithms and makes heavy use of the two-language paradigm. For this reason I implemented the source code of *rdist* separately in Cython[15] [Behnel et al., 2011]. Cython is a language extension of Python, which is compatible with Python but at the same time provides features of compiled languages. Choosing to use another programming language solves various issues for *rdist* when used in combination with *datafold*:

- **Make use of high-performance libraries:** In the second step of Algorithm 1, I require an efficient state-of-the-art implementation of a search algorithm. Cython allows direct and seamless interfacing with high-performance libraries such as from C, C++ or Fortran. It is possible to translate between standard *NumPy* arrays and C-style pointer variables such that data can be passed from Python to a compiled language with zero copy.

- **Maintain full compatibility to Python:** After the code is compiled, Cython bundles the software to a normal Python package. This means that *rdist* can be included in *datafold* as a regular package and distance matrix algorithm (cf. Fig. 3.7).

  The compatibility also works in the other direction: Cython allows importing Python packages. I can therefore still use the same packages from the scientific computing stack. In *rdist* I make use of this feature in the "project" step, where I use *scikit-learn* to perform the PCA, and in the "pullback" step where I can directly return the final distance matrix in a compressed sparse row format from the *SciPy* package.

- **Remove restrictions from the Python interpreter:** As highlighted in Section 2.5.2, Python is well-suited for a clear and flexible programming front end. However, it comes with serious computational limitations because of the interpreter process and the Global Interpreter Lock (GIL). In Cython the GIL can be explicitly disabled, which gives access to shared memory parallelization (e.g. by using OpenMP[16]). I utilized this feature for the "pullback" operation in the last step of Algorithm 1, which is an embarrassingly parallel operation:

```
# loop over all canidate pairs (i,j) in parallel and compute
# the pairwise distance in the ambient space:
for k in prange(
            n_candidates, schedule="dynamic", nogil=True):
    i, j = candinate_set[k]
    dist[k] = compute_distance(X[i, :], X[j, :])
```

[15] https://cython.org/
[16] https://www.openmp.org/

The `prange` acts as a typical Python built-in `range` operation but performs the loop in parallel. The code snippet is a simplified version of the actual code in *rdist*.

In a benchmark analysis in Section 4.4 I include both simulated and real-world datasets. In the final configuration of *rdist*, I use a kd-tree of the C++ library *mlpack* [Curtin et al., 2018]. This choice is the result of the benchmark analysis, where I found that *mlpack* generally provides the fastest state-of-the-art implementations of tree-based algorithms to construct sparse distance matrices. The success of the "project-search-pullback" idea is that *rdist* together with the kd-tree of *mlpack* outperforms the kd-tree used on its own.

3.4 *dynfold*: State representation and linear system identification

The middle layer of *datafold's* software architecture includes two types of data-driven methods: data transformations to obtain a new state representation (which can be can also be split into temporal or spatial feature extraction methods), and regressions of time series data for system identification. The approximation of operators becomes essential for both types of methods. Mathematically, the two types are described by deriving the following functions

$$\mathbf{g}(\mathbf{x}_j) = \mathbf{z}_j \qquad \mathbf{g} : \mathbb{R}^N \to \mathbb{R}^M \qquad \text{state representation} \tag{3.4}$$

$$\mathbf{x}_{j+1} = F_{\Delta t}(\mathbf{x}_j) \qquad F_{\Delta t} : \mathbb{R}^N \to \mathbb{R}^N \qquad \text{system identification} \tag{3.5}$$

The (unknown) function $\mathbf{g}(\mathbf{x})$ describes a coordinate change of the data $\mathbf{x}$ to a new state representation $\mathbf{z}$. In a machine learning perspective, this corresponds to an unsupervised task, which means that the target values $\mathbf{z}$ are not explicitly given and are instead extracted from the data. In Eq. 3.5, $F_{\Delta t}$ describes the flow of a dynamical system with a constant time increment of Δt between the states. This corresponds to a supervised regression task, which is imposed by the state evolution in the time series. Importantly, at *dynfold* all methods that solve either of the two tasks in Eq. 3.4 – 3.5 are treated separately to promote modularity within the layer. A combination of methods is then covered in the higher level *appfold*, which I describe in the next section.

Fig. 3.8 includes the main class organization in *dynfold* together with the three main methodological components of this thesis: (1) time delay embedding to extract temporal features (Eq. 3.4), (2) DMAP to extract spatial features (Eq. 3.4) and (3) DMD for mode decomposition (Eq. 3.5). I describe each of the methodological components in Sections 3.4.2 – 3.4.4. In the next section I highlight the class organization of data-driven methods, which perform the two tasks of Eq. 3.4 – 3.5.

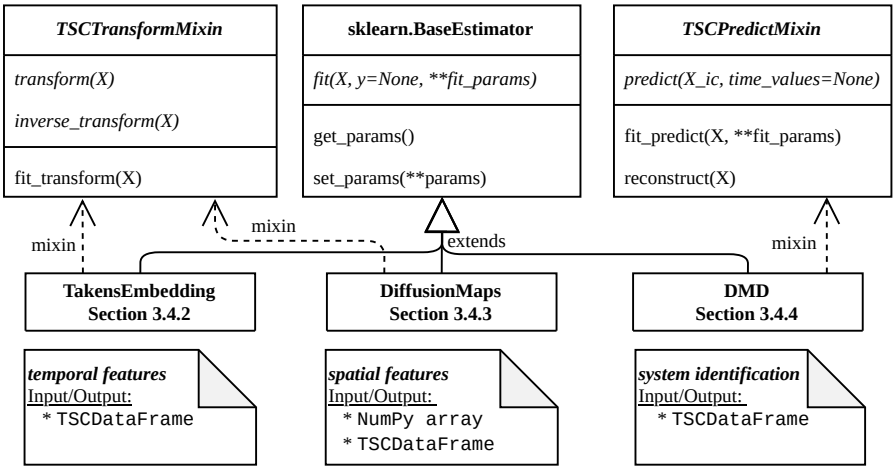


Figure 3.8: The class organization in *dynfold* mirrored from *scikit-learn*. All classes that solve data-driven tasks inherit from `BaseEstimator` and mixins according to which task is addressed. Attributes in italics are abstract.

3.4.1 Mixin design pattern to organize data-driven methods

According to *datafold's* design decision, the application programming interface (API) of all methods in *dynfold* align to *scikit-learn* [Buitinck et al., 2013; Pedregosa et al., 2011]. While the *scikit-learn* package already covers common tasks such as feature extraction in Eq. 3.4 or function regression in Eq. 3.5, a main objective in this section is to extend the class organization and API for time series collection data (`TSCDataFrame`) and system identification. A goal is to remain as compatible as possible with *scikit-learn* to make use of its available functionality.

To achieve this, the base class of all data-driven methods is `BaseEstimator` from *scikit-learn*, irrespective of the actual task. Because of the generality of the base class, it does not include any specific functionality and only performs rudimentary checks of whether an inheriting class follows the design rules of *scikit-learn*[17]. `BaseEstimator` requires that an inheriting subclass implements a `fit(X)` method. This performs the training of a model with data `X` in the argument.

A principle design pattern in *scikit-learn* — which I mirror in *datafold* — are so-called "mixin classes" (also referred to as "refinement classes"). Mixins are an approach in software engineering in which a class encapsulates reusable code fragments that can be integrated into independently-defined classes [Bracha and Cook, 1990; Smaragdakis and Batory, 2002]. A mixin-based software design promotes a form of "incremental programming", in which building blocks can be added to classes that require the same functionality [Bracha and Cook, 1990].

Because the integration is often performed by class inheritance, the software pattern is best practiced in object-oriented programming languages that support multiple class

[17] https://scikit-learn.org/stable/developers/develop.html

inheritance [Bracha and Cook, 1990], which is the case in Python. The class inheritance is formulated as a composition of mixins and provides a compromise between "true multi inheritance" (in a strict sense) and restrictive single inheritance, such as in the Java programming language [Smaragdakis and Batory, 2002].

In *scikit-learn*, mixins are used to distinguish between learning tasks and attach common task-specific code to a class by inheritance. The schematic layout of a data-driven method that integrates two mixins is:

```
class MethodImpl(BaseEstimator, Mixin1, Mixin2):
      [...]
```

In this case `MethodImpl` "is-a" `BaseEstimator` and integrates two kinds of functionality from `Mixin1` and `Mixin2`. The *dynfold* layer introduces two mixins, which correspond to the two tasks of state representation and system identification:

`TSCTransformMixin` covering tasks in Eq. 3.4:
The class `TSCTransformMixin` generalizes *scikit-learn's* `TransformMixin` to support both static and temporal data. The functions that should be available in a "transformer class" are `transform(X)` and `inverse_transform(X)` (only if the inverse exists). These functions are identical to the existing `TransformMixin` and therefore fully compatible to *scikit-learn*. However, the prefix "TSC" highlights that the mixin includes additional functionality to process and validate `TSCDataFrame` data and maintain the time indices in the data structure to the output. In addition to *spatial* feature extraction methods, the new `TSCTransformMixin` therefore enables *temporal* feature extraction, such as time delay embedding in `TakensEmbedding` (Section 3.4.2), which is currently not possible in the standard *scikit-learn*.

While Fig. 3.8 includes the main data transformations used in this thesis, Table 3.3 lists a set of standard methods that are included in *dynfold*. All classes with dependency use existing and well-tested implementations and require only minor adaptations to process the data structure `TSCDataFrame`, which is a benefit from the compatible data layout of `TSCDataFrame` (cf. Table 3.2).

Table 3.3: Standard data transformations to extract spatial (s) or temporal (t) features. All classes are included in the layer *dynfold*.

Class	Description	s/t	Dependency
`FeaturePreprocess`	Wrapper for basic pre-processing, such as data normalization	s/t	*scikit-learn*[1]
`Identity`	Pass-through data; well-suited for testing.	s/t	-
`PrincipalComponent`	Linear projection on principal components.	s/t	*scikit-learn*[1]
`RadialBasis`	Describe data in coefficients of radial basis functions.	s/t	*SciPy*[2]
`PolynomialFeatures`	Generate polynomial combinations of data.	s/t	*scikit-learn*[1]
`ApplyLambda`	Element-wise data transformations as specified by user	s/t	-
`FiniteDifference`	Approximate time derivative from data with difference schemes.	t	*findiff*[3]

[1]Pedregosa et al. [2011]; [2]Virtanen et al. [2020]; [3]Baer [2021]

`TSCPredictMixin` covering tasks in Eq. 3.5:
The `TSCPredictMixin` is utilized in methods that perform system identification from time series data and can predict future states according to the identified flow ($F_{\Delta t}$ in Eq. 3.5). This poses a multivariate regression problem (assuming the common case of $N > 1$). `TSCPredictMixin` therefore templates from the API for supervised tasks. The main function `predict(X)` maps an input state to future system states. There is a key difference compared to regression methods of static data. Because of the recursive nature of a system's flow $F_{\Delta t}$, the time series data X already includes the snapshot pairs that provide the input and target states, $(\mathbf{x}_j, \mathbf{x}_{j+1})$. This means a single sample can serve both as input and target vector, while in static data these are more separated. Moreover, applying a flow for a larger prediction horizon than Δt means that a single initial sample can map to many future states. Unlike a one-to-one map in static regression, the map is one-to-many.

The class `TSCPredictMixin` also adds a new attribute `reconstruct(X)` which does not exist in the *scikit-learn* API. The function is meant to reconstruct time series collection data by extracting the initial states from all time series in a `TSCDataFrame` (usually containing true measurements) and evaluate the model at the same time values of the time series. This operation turned out to be practical for model validation and comparing observed and predicted data (cf. Section 3.3.1.3).

3.4.2 Time delay embedding

Time delay embedding is a method to deal with ill-defined dynamics. During the embedding previous states are augmented to a new state which can reconstruct the state dynamics (background in Section 2.4.3). The method classifies as temporal feature extraction and therefore exclusively operates on time series data. The following code snippet provides a schematic layout of the class to perform time delay embedding in *datafold*:

```
class TakensEmbedding(BaseEstimator, TSCTransformMixin):
    def __init__(self, delays, kappa):
        # set parameters in Eq. 2.44

    def fit(self, X: TSCDataFrame):
        # investigate input X and set internal attributes

    def transform(self, X: TSCDataFrame):
        # inherited from TSCTransformMixin
        self._validate_data(X, min_length=self.delays+1,
                            delta_time=delta_time_fit)

        # perform data transformation in Eq. 2.44
        for ts in X.itertimeseries():
            # perform time delay embedding
        return X_embedded

    def inverse_transform(self, X: TSCDataFrame):
        # only return the original feature names
        return X.loc[:, self.feature_names_in_]
```

The inheritance profile corresponds to the diagram in Fig. 3.8, where the base class `BaseEstimator` and mixin `TSCTransformMixin` require implementing the `fit(X)`, `transform(X)` and `inverse_transform(X)` method. The transformation is straightforward and can be easily performed in a per time series fashion by using the iterators of `TSCDataFrame` (`X.itertimeseries()`). The inverse transformation is trivial in that it only returns the original states.

The code snippet also exemplifies the use of common functionality that is included with `TSCTransformMixin`. In the `transform(X)` method, the data in `X` is validated in `_validate_data(...)`. The statement checks whether the data contain the same feature names and have the same sampling rate as during training (in `fit(X)`). Furthermore, all time series must contain an equal or greater number of states than the specified `delay+1` parameter to perform the embedding.

An important side-effect of time delay embedding is that it increases the point dimension ($M > N$) but reduces the overall number of samples in a time series. A single

target vector has a dimension of $M = N(d+1)$, where d is the number of delays. The first d states of each time series therefore have no corresponding output state. This also means that in a prediction task to perform the embedding, $\mathbf{g}(X_{d+1}) = \mathbf{z}$, an initial state now requires a time series X with $d+1$ measurement samples.

3.4.3 Diffusion Maps

Diffusion Maps (DMAP) is a versatile and widely-adopted kernel-based method in data-driven research [Coifman and Lafon, 2006b]. A primary focus of the method is to robustly extract interesting geometric structures from point clouds. The method has been used for static data [e.g. Coifman and Lafon, 2006b] and has been frequently integrated for the analysis and modeling of temporal data [e.g. Berry et al., 2013; Dietrich et al., 2016; Giannakis, 2019; Mauroy et al., 2020].

In the main setting of this thesis, as displayed in Fig. 3.1, DMAP is a central component for operator-informed system identification. In the setting, I make use of DMAP to approximate the eigenfunctions of the Laplace-Beltrami operator on time series data (i.e. potentially time-delayed as in Fig. 3.1). The obtained eigenfunctions serve two purposes: to extract latent manifold coordinates that relate to the hidden state space — with the assumption of a compact Riemannian state space manifold — and to obtain a geometric aligned function basis to approximate the Koopman operator.

The main requirements of a method implementation to capture these tasks within my thesis are

1. support of time series collection data
2. compatibility with the *scikit-learn* API
3. support of an out-of-sample extension
4. support of a sparse kernel matrix

These requirements are summarized in Table 3.4. Moreover, the table includes three additional features, which I have integrated into my own implementation:

5. support of an arbitrary kernel
6. conjugate transformation of the kernel matrix to increase numerical stability
7. arbitrary eigenproblem solver for algorithmic experimentation

Despite the importance of DMAP in research, I found that openly available software packages lack essential functionality; see Table 3.4. As one of the main packages for machine learning in Python, the *scikit-learn* project already provides a variety of manifold learning algorithms. However, it only includes "spectral embedding" as a special case of DMAP in which the characteristic normalization steps are not included ($\alpha = 0$

in Eq. 2.38 on page 45). Consequently, the eigenfunctions of the Laplace-Beltrami operator are only well-approximated if the points in a dataset are uniformly distributed on the underlying manifold, which is rarely the case in practice [Coifman and Lafon, 2006b].

Table 3.4: Comparison of Python packages that implement the Diffusion Maps method (Abbreviations: arbit.=arbitrary, conjug.=conjugate, compat.=compatibility oos=out-of-sample, eig.=eigenvalue).

Package	time series	sklearn compat.	oos	sparse kernel	arbit. kernel	conjug. kernel	arbit. eig.solver
PydiffMap[1]	no	yes	yes	k	no	no	no
scikit-learn[2]	no	yes	no	k	no	no	yes
DiffusionMaps[3]	no	no	no	δ	no	no	no
megaman[4]	no	no	yes	k	no	no	yes
UQPy[5]	no	no	no	k	yes	no	no
`DiffusionMaps`[6]	yes	yes	yes	δ	yes	yes	yes

[1]Thiede et al. [2021]; [2] Pedregosa et al. [2011]; [3]Bello-Rivas [2017]; [4]McQueen et al. [2016]; [5]Olivier et al. [2020]; [6]*datafold*

Fig. 3.9 gives an overview of the software design of the Python class `DiffusionMaps`. The mathematical equations for DMAP are covered in Eq. 2.37 – 2.44 on page 45. According to the class hierarchy established in *dynfold*, as an unsupervised spatial feature extraction method `DiffusionMaps` inherits from `BaseEstimator` and includes functionality of `TSCTransformMixin`.

Much of the necessary flexibility in `DiffusionMaps` is already established in the previous *pcfold* layer, which covers time series collection and an interface to compute a (dense or sparse) distance matrix. The kernel function is simply applied on a given pairwise distance matrix (Eq. 2.36 on page 40). The kernel interface is similar to the distance matrix in that it has a single main function `__call__(X, Y)` at which the kernel is computed. Both `X` and `Y` can be either static (`ndarray`) or temporal data (`TSCDataFrame`). The second argument distinguishes how the kernel is computed: pairwise during training (`Y=None`) and component-wise for out-of-sample evaluations (`Y` is given).

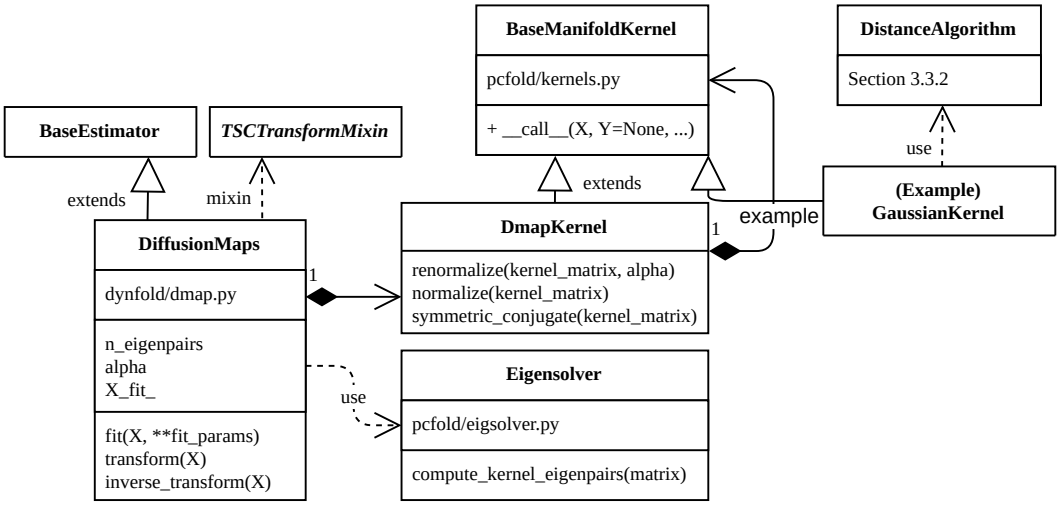


Figure 3.9: Class diagram for `DiffusionMaps` class. The kernel interface and eigensolver selection is covered in *pcfold*. In the example a `GaussianKernel` is used within the dedicated `DmapKernel`, which performs the characteristic normalization of a kernel matrix to describe the diffusion process.

A main design idea in the software layout of `DiffusionMaps` is that the normalization steps and matrix conjugation (Eq. 2.38 – 2.39) are encapsulated in a dedicated "meta-kernel" class `DmapKernel`; Fig. 3.9. This means that the symmetric and positive kernel, which describes the point similarity (in Fig. 3.9 the `GaussianKernel` is shown as the default) is wrapped in `DmapKernel` as another kernel. This provides great flexibility because the three operations are now separated: (1) distance matrix, (2) kernel and (3) DMAP specific normalization/conjugation steps. Ultimately, this allows a user to arbitrarily specify the distance metric and kernel. Surprisingly, only one (*UQPy*) of the other packages in Table 3.9 covers this feature, despite modifications at the kernel specification being the main source of methodological variation in the DMAP framework. The following list highlights available kernels from recent research that are included in *pcfold*. Detailed explanations are left to the cited articles as this is beyond the scope of my thesis.

- Various radial basis kernels, such as the Gaussian kernel as the default [Coifman and Lafon, 2006b].
- Continuous k-nearest neighbor kernel (`ContinuousNNKernel`); computes an *unweighted* kernel matrix, which comes with additional guarantees for topological data analysis [Berry and Sauer, 2019].
- A `ConeKernel` as a "dynamically-adapted kernel", which integrates the temporal context in the point proximity by using finite difference schemes [Giannakis, 2015; Giannakis and Majda, 2013].

Dynamically-adapted kernels are particularly interesting for my thesis because they include the temporal context of time series data. In fact, the composed data transformation of time-delayed states and subsequent kernel function is also characterized as a dynamically-adapted kernel [Giannakis, 2015]. Note that in the comparison of Table 3.4, `DiffusionMaps` is the only class that can process time series data and therefore the only implementation that supports dynamically-adapted kernels.

The following code snippet highlights a typical workflow that follows from the available API in `TSCTransformMixin` (cf. Fig. 3.8):

```
X = make_swiss_roll(5000)          # data to fit model
X_oos = make_swiss_roll(250)       # out-of-sample data

# set up model
dmap = DiffusionMaps(
            # insert arbitrary kernel
            kernel=GaussianKernel(6),
            # number of eigenpairs to compute
            n_eigenpairs=7,
            # distance matrix specification
            dist_kwargs={"cut_off": 3, }
)

# construct model and map data new state representation
Psi = dmap.fit_transform(X)
# map out-of-sample data to the new state representation
Psi_oos = dmap.transform(X_oos)
```

The code listing uses data sampled from the swiss-roll manifold, which is also used as an example geometry in Section 2.4.2. Fig. 3.10 displays the coordinates (ψ_2, ψ_6), in which the swiss-roll is embedded into a two dimensional space that keeps important geometric information of the swiss-roll.

An essential feature for the map is the out-of-sample extension, in which the eigenvectors as point evaluations are expanded to the neighborhood region. The out-of-sample extension is required if the DMAP is integrated into a supervised task, such as system identification. The standard method in `DiffusionMaps` is the Nyström extension, which is also included in Eq. 2.42 – 2.43 on page 45. The Nyström extension requires the component-wise evaluation of internal normalized kernel (i.e. $\mathcal{K}(X, Y)$, where Y includes the out-of-sample data). The availability of the component-wise evaluation of a *normalized* kernel highlights the advantage of a dedicated meta-kernel `DmapKernel`. An example of the out-of-sample mapping for the swiss-roll is given in Fig. 3.10 where the separate test samples in the right plot have the same mapping as in the identified eigenmap in the left plot (indicated by the color code).

The *dynfold* layer also contains implementations of more involved out-of-sample extensions, such as a multi-scale Laplacian pyramid [Fernández et al., 2020] and geomet-

ric harmonics [Coifman and Lafon, 2006a]. However, because these methods require solving a separate regression, I do not detail the additional extensions. In the data analysis in the next chapter, I only use the Nyström extension.

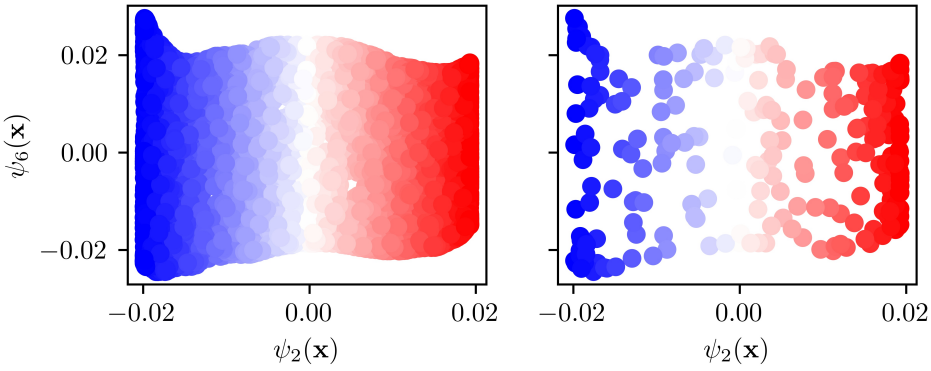


Figure 3.10: Unrolling the swiss-roll in two coordinates. Left: Embedding of the training data with 5000 samples onto the second and sixth eigenfunction. Right: Out-of-sample mapping of 250 samples to the same coordinates that are not in the training data.

The last feature in Table 3.4 describes an ability to choose which eigensolver to use. This is relevant because computing the eigenpairs (Eq. 2.40) often represents the computational bottleneck. Depending on the kernel and model parametrization different eigensolvers can be suitable. Typically, only the leading eigenpairs are computed from the (normalized) kernel matrix (parameter `n_eigenpairs`). In the standard implementation, this requires iterative algorithms that handle a matrix that is non-symmetric because of the row-stochastic normalization step in DMAP. However, if the symmetric conjugation of the matrix (Eq. 2.39) is performed, it is possible to use dedicated eigensolvers for symmetric matrices. Overall this improves the numerical stability and computational efficiency, particularly if the matrix is sparse. Depending on the setting and resulting matrix properties, the default behavior is that the eigensolver backend is selected from the *SciPy* package `eigs` or `eigsh`, which utilize Arnoldi iterations [Virtanen et al., 2020].

The flexible choice of eigensolver permits algorithmic experimentation to accelerate the computation. A student group at the Technical University of Munich used this flexibility to compare the default *SciPy* eigensolvers with an external C++ library *SLEPc* (Scalable Library for Eigenvalue Problem Computations) [Hernandez et al., 2009]. They could show that it is possible to delegate the iterative computations to a Linux cluster (using up to 256 cores) and that for larger datasets *SLEPc* outperforms the default *ARPACK* solvers by a factor of about two. The setup and results are contained in an openly available technical report [Grad and Raith, 2021].

3.4.4 Dynamic Mode Decomposition

In this section, I continue with variants of the Dynamic Mode Decomposition (DMD) intending to perform system identification. The assumption for the basic DMD is that the states in the time series are well-defined and can be described with linear dynamics. These assumptions are rarely fulfilled in practice. The main objective is to encapsulate DMD variants as a modular component as part of the broader Extended Dynamic Mode Decomposition (EDMD) to perform nonlinear system identification in Section 3.5.

Section 2.3.2 describes DMD as a powerful methodology. At its core the most basic DMD variant first performs a linear regression to solve for a system matrix $U_{\Delta t}$ and subsequently diagonalizes this matrix into spectral components:

$$U_{\Delta t} = X_+ X_-^{\dagger} \quad (3.6)$$

$$U_{\Delta t} = \Phi \Lambda \Phi^{-1} \quad (3.7)$$

```
X: TSCDataFrame
X_m, X_p = X.tsc.shift_matices(X)
U = np.linalg.lstsq(X_m, X_p).T
Phi, Lambda, Phi_i = diagonalize(U)
```

The left side shows the central equations and the right side the corresponding code by using available functionality of the ***pcfold*** layer. Note that the mathematical equations are expressed in column-oriented states, whereas the states in `TSCDataFrame` are row-oriented (following the convention of ***scikit-learn***).

The two matrices X_+ and X_- are obtained by shifting all time series within the time series collection. The corresponding operation `shift_matrices` is provided in the accessor of `TSCDataframe` (see Section 3.3.1.2). For simplicity I assume that the matrix $U_{\Delta t}$ has full rank for the subsequent diagonalization in Eq. 3.7.

The matrix $U_{\Delta t}$ and the spectral components $(\Phi, \Lambda, \Phi^{-1})$ can either be used to analyze the estimated dynamical system, or to perform state interpolations or future predictions. In the class organization of ***dynfold*** this corresponds to a (supervised) system identification task as per Eq. 3.5.

Despite its high theoretical value as a linear decomposition method, openly available software that provides a well-structured interface to utilize the mode decomposition for system identification is scarce. The largest Python package providing a software structure and implementation of various DMD variants is ***PyDMD*** [Demo et al., 2018]. Similar to ***datafold***, the ***PyDMD*** package is compatible with ***scikit-learn*** and builds on the scientific computing stack of Python (cf. Fig. 2.10). However, it comes with two significant shortcomings for the requirements of my thesis:

1. Only single and coherent time series are supported for the input data. This counteracts the established data structure `TSCDataFrame` to support generalized sampling schemes of dynamical systems. In the literature these generalizations are described for the DMD in Tu et al. [2014]. All of my data analysis in Chapter 4 requires this generalization to time series *collection* data.

2. ***PyDMD*** does not make use of the full capabilities of the underlying linear dynamical system. For example, it is not possible to interpolate states below the time

sampling rate Δt or predict the full solution trajectory based on a single initial condition.

An additional non-essential feature missing from ***PyDMD*** is that it does not support approximating a differential form of the dynamics. To overcome these limitations I describe a software design to organize standard DMD variants in ***dynfold***. This is illustrated in a class diagram of Fig. 3.11. Currently, there are three DMD variants and a wrapper for the DMD variants in ***PyDMD***:

`DMDFull`	full decomposition as per Eq. 3.7
`DMDEco`	perform an "economical" DMD that linearly reduces the states with a Singular Value Decomposition (SVD). This is useful if the original measurements $\mathbf{x} \in \mathbb{R}^N$ are high dimensional $N \gg 1$ [e.g. Kutz et al., 2016a]
`gDMDFull`	full decomposition of the system generator $L = \dot{X}_+ X_-^\dagger$, where $\dot{X}$ contains the time derivatives of the time series [Klus et al., 2020]
`PyDMDWrapper`	access to the various DMD variants in ***PyDMD*** such as the higher-order DMD [Le Clainche et al., 2017], multi-resolution DMD [Kutz et al., 2016b] or forward-backward DMD [Dawson, 2016]

An advantage of having the wrapper class of ***PyDMD*** is that it gives access to the additional DMD variants. These, however, come with the above-mentioned shortcomings. I mainly use the wrapper to cross-check the implementations contained in ***datafold's*** unit tests.

As shown in Fig. 3.11, all DMD classes have `DMDBase` as the only base class. `DMDBase` already provides implementations for `predict`, `reconstruct` and `score` that are shared by all concrete sub-classes. Therefore, a new DMD class only requires providing the `fit(X)` method in which the decomposition of the time series data in X is carried out. Ultimately, this is where the Koopman operator is approximated in a matrix form. Overall, this allows an easy integration of new DMD variants and acknowledges the active research in the field (cf. Table 2.3).

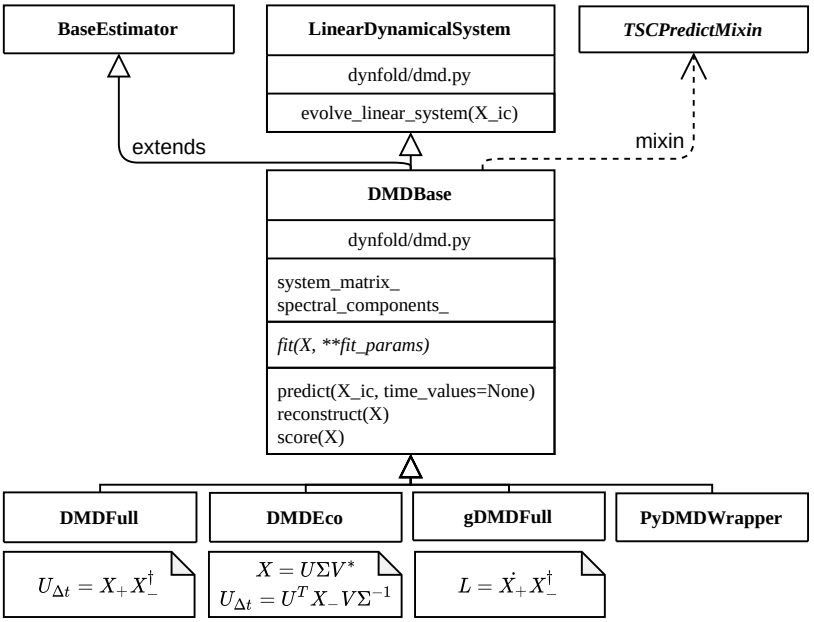


Figure 3.11: Class diagram for DMD variants in *dynfold*. The classes `DMDBase` and `LinearDynamicalSystem` include functionality that are common for all variants. As a method for system identification the `DMDBase` (and its variants) inherits from `BaseEstimator` and `TSCPredictMixin`.

`DMDBase` itself inherits from the standard class `BaseEstimator` for data-driven models and `TSCPredictionMixin` to highlight its usage for system identification. Within `fit` each DMD method is responsible for setting up attributes to set up a linear system within the inherited class `LinearDynamicalSystem`. These attributes can be (1) the original system matrix $U_{\Delta t}$ in Eq. 3.6 or (2) its spectral components $(\Phi, \Lambda, \Phi^{-1})$ in Eq. 3.7. Table 3.5 highlights these two cases for the flow. Similarly a DMD class can also describe the system in a differential form with a generator matrix L. Note that the generator has the same eigenvectors, but different set of eigenvalues $\Omega = {}^{\log(\Lambda)}/_{\Delta t}$ (for details see Section 2.1). In principle each case is equivalent and predicts the same solution trajectory — because of the linearity each solution exists and is unique for any given initial condition $\mathbf{x}_1$ or $\mathbf{b}_1 = \Phi^{-1}\mathbf{x}_1$. However, there are analytical and computational differences. The spectral system form has a higher analytical value because it describes the system in intrinsic components. Furthermore, the system evaluates more efficiently because the time dependent matrices Λ and Ω are diagonal. Nevertheless, setting up a system with the matrix $U_{\Delta t}$ can be beneficial to analyze structural information in the matrix and also avoids the diagonalization.

Table 3.5: Variants to evaluate the flow of a continuous and time invariant linear dynamical system. The exp denotes the matrix exponential.

Evaluation	**Flow** ($U_{\Delta t} = \Phi \cdot \Lambda \cdot \Phi^{-1}$)	**Vector field** ($L = \Phi \cdot \Omega \cdot \Phi^{-1}$)
system matrix	$\mathbf{x}_t = U_{\Delta t}^{t/\Delta t} \cdot \mathbf{x}_1$	$\mathbf{x}_t = \exp(L \cdot t) \cdot \mathbf{x}_1$
spectral components	$\mathbf{x}_t = \Phi \cdot \Lambda_{\Delta t}^{t/\Delta t} \cdot \mathbf{b}_1$	$\mathbf{x}_t = \exp(\Omega \cdot t) \cdot \mathbf{b}_1$

The following code highlights a typical system identification workflow with `DMDFull`:

```
X: TSCDataFrame          # measurement data
X_ic: TSCDataFrame       # initial conditions
metric: TSCMetric        # specified metric (Section 3.3.1.3)
score: TSCScoring        # specified score (Section 3.3.1.3)

# perform decomposition and specify linear system
dmd = DMDFull().fit(X)

# predict time series for each initial condition in X_ic
X_est = dmd.predict(X_ic, time_values=np.linspace(0, 5, 10))

# reconstruct the original measurement data
X_reconstruct = dmd.reconstruct(X)

# investigate model behavior and error
metric(X, X_reconstruct)
score(X, X_reconstruct)
```

During `fit(X)`, the method decomposes the time series according to Eq. 3.6 – 3.7 and sets the necessary attributes for a case in Table 3.5 (defaulting to the flow in spectral form). The model's function `predict(X_ic, time_values)` is used to evaluate multiple initial conditions in `X_ic`. The returned data type is a `TSCDataFrame`, which contains the solution time series for each initial condition in `X_ic`. Each time series is evaluated at the `time_values` given in the second argument — a sorted array of arbitrary many numbers (real and positive).

From the specification of `time_values` follows that the requested state evaluations are independent of the original sampling rate Δt. From a mathematical and numerical treatment this can pose a problem: the exponentiation $\lambda^{t/\Delta t}$ can have a real-valued exponent and complex-valued base number. Generally, this is a multi-valued function, that is, the operation can have multiple solutions. To resolve this situation, I use the so-called *principal value* as a strategy to always pick a certain solution, which makes the operation again well-defined. I rely on the standard strategy implemented in *NumPy*, which I found suitable to interpolate time-continuous dynamical systems. For details on this issue see Mauroy and Goncalves [2020] or Dietrich et al. [2020].

The `reconstruct` function is introduced in `TSCPredictMixin` for system identification tasks and is not available in the standard *scikit-learn* API. In the function, a time series collection `X` is reconstructed by evaluating the model at identical time values and the initial condition for each time series in `X`. This makes it easy to validate the model on available data with a metric and score (see the last two statements in the above code snippet). As highlighted in Section 3.3.1.3, this can give valuable information about the model's predictive capabilities and can help investigate the structural error in the model to guide the model revision in the system identification loop.

In the next section, the capabilities of DMD are extended to nonlinear system identification by making use of all of the established methods.

3.5 *appfold*: Nonlinear system identification

The third and highest layer *appfold* is dedicated to methods that solve complex data-driven applications. Typically, the methods in this layer require the functionality of the previous two layers. The main focus of my thesis is to provide a generic software design for the EDMD framework to approximate the Koopman operator and its spectral components for system identification. While I focus on descriptive and forecasting tasks, the *appfold* layer is also intended for methods that can solve related tasks. For example, Surana [2020] describes an interdisciplinary framework based on the Koopman operator, which also includes time series classification or anomaly detection tasks. But methods for model predictive control [Mauroy et al., 2020] or analog forecasting [Alexander and Giannakis, 2020] could also be integrated in future development.

In Section 3.5.1, I first describe `EDMD` as the central class in this third layer. Moreover, in Section 3.5.2 I contribute a class with which it is possible to optimize the parameters contained in a `EDMD` model by minimizing a cross-validated error. This finalizes the entire system identification loop with a model architecture based on the Koopman operator. Finally, I compare the features of my proposed `EDMD` class to other software projects in Section 3.5.3.

3.5.1 Extended Dynamic Mode Decomposition

As detailed in Section 2.3, the main objective of EDMD is to compute the Koopman matrix from time series data and spectrally decompose the matrix into the Koopman triplet — the modes V, eigenvalues $\Lambda_{\Delta t}$ and eigenfunctions $\xi(\mathbf{x})$. These components define the final model

$$\mathbf{x}_{j+1} = V\Lambda_{\Delta t}^{j}\xi(\mathbf{x}_1) \tag{3.8}$$

which despite being linear is also capable of describing nonlinear state evolution in the measurement states $\mathbf{x}$. This is possible because the system operates on a different state representation, indicated by the Koopman eigenfunctions, $\xi(\mathbf{x}) = \Phi^{-1}\mathbf{g}(\mathbf{x})$. The $\mathbf{g}(\mathbf{x})$ are the intrinsic states of the EDMD dictionary and Φ^{-1} are the left eigenvectors of the Koopman matrix $U_{\Delta t}$. The Koopman modes V (linearly) reconstruct the original

measurements from the intrinsic states. For the derivation of the model components see Eq. 2.26 – 2.32 on page 34.

The objective of the `EDMD` class is to manage the two main tasks: (1) provide a generic and flexible dictionary $\mathbf{g}(\mathbf{x})$ and (2) compute and store the Koopman triplet. Together this leads to a dynamical system model in Eq. 3.8 as an operator-based approach to perform nonlinear system identification. My goal for the design of `EDMD` is to reuse existing functionality from the previous layers as well as within *datafold's* dependencies to the scientific computing stack.

Table 3.6 compares different aspects of my proposed `EDMD` implementation in *appfold* against the original formulation in Williams et al. [2015]. The aspects show that in the software design, I continue to support time series collection data. Moreover, I also generalize two aspects concerning the dictionary and the mode decomposition.

Table 3.6: Comparison of the original formulation of EDMD in Williams et al. [2015] to the `EDMD` class in the layer *appfold*.

Aspect	**Original EDMD**	**`EDMD` class**
data format	time series collection	time series collection
dictionary	spatial	spatial and/or temporal
mode decomposition	full DMD	arbitrary DMD variant from `DMDBase`

Besides the time series data itself, an essential element of EDMD is the dictionary $\mathbf{g}(\mathbf{x}) = \mathbf{z}$ (Eq. 2.26). The dictionary corresponds to the "extended part" in EDMD and describes a data transformation of the original measurements to a new state representation $\mathbf{z}$. In particular, the objective given by the Koopman operator theory is to select a dictionary that (approximately) linearizes the state dynamics. However, a suitable dictionary for a concrete system depends on many often unknown factors. The dictionary choice is therefore regarded as an "open problem". Within a software solution, it is an important feature to have a flexible choice of the dictionary to be able to test multiple configurations.

A limitation of the original EDMD is that it only describes spatial transformations within the dictionary (Williams et al. [2015] use Hermitian or radial basis functions). On the other side, there are DMD variants that generalize this aspect to temporal feature extraction, such as the Hankel-DMD [Arbabi and Mezić, 2017]. However, these variants are again not equipped with a dictionary that would allow including further spatial transformations. In my `EDMD` solution I support chaining multiple methods, which can be both spatial and temporal feature extraction methods. This is particularly relevant for the main setting of my thesis in Section 3.1, where I include both time delay embedding (temporal) and Diffusion Maps (spatial).

Another generalization that I perform is in the "DMD part" of EDMD. While the original formulation describes a full decomposition (as stated in Eq. 3.7), I make use of available DMD variants organized in `DMDBase` within the previous *dynfold* layer. This

means the actual mode decomposition to approximate the Koopman components can be performed by an arbitrary DMD variant. For example, this allows approximating the Koopman *generator* by setting `gDMDFull` leading to gEDMD as described in Klus et al. [2020].

In summary, the `EDMD` class consists of two parts: one or many methods to describe the dictionary and a method to perform the final mode decomposition. This is highlighted in the class diagram of Fig. 3.12. To specify each of the parts, I can make use of the functionality of the previous ***dynfold*** layer. Ultimately, the `EDMD` can be interpreted as a "meta-model" because it organizes available data-driven methods into a single method.

To achieve this proposed flexibility in `EDMD`, I make use of the existing functionality of ***scikit-learn***. The main idea is to use the `Pipeline` class as a base class of `EDMD`. According to the ***scikit-learn*** documentation[18] (version 1.0.1): "A Pipeline can be used to chain multiple estimators into one. [...] All estimators in a pipeline except the last one, must be transformers."

A pipeline therefore reflects a typical pattern in machine learning, which I transfer to the `EDMD` class: Chain one or multiple feature extraction methods (the EDMD dictionary) and then pass the time series to a final estimator — the linear system identification in the dictionary space. Contrary to the `Pipeline`, `EDMD` exclusively processes time series data of type `TSCDataFrame`. Furthermore, the class includes the additional procedures to compute and store the Koopman triplet to set up the final system in Eq. 3.8.

As shown in Fig. 3.12 the `EDMD` has *two* mixin classes — `TSCTransformMixin` and `TSCPredictMixin`. It therefore provides both of the main functionalities. The `transform(X)` performs the map described by the EDMD dictionary $\mathbf{g}(\mathbf{x})$ on time series, whereas `inverse_transform(X)` linearly reconstructs the time series in the dictionary space back to the original measurement states (matrix B in Eq. 2.27) (Note that the Koopman modes reconstruct the Koopman eigenfunctions.) Like for a standard DMD variants in Section 3.4.4, the method `predict(X_ic, time_values)` returns a time series prediction for each initial condition in `X_ic` (as per Eq. 3.8).

[18] https://scikit-learn.org/

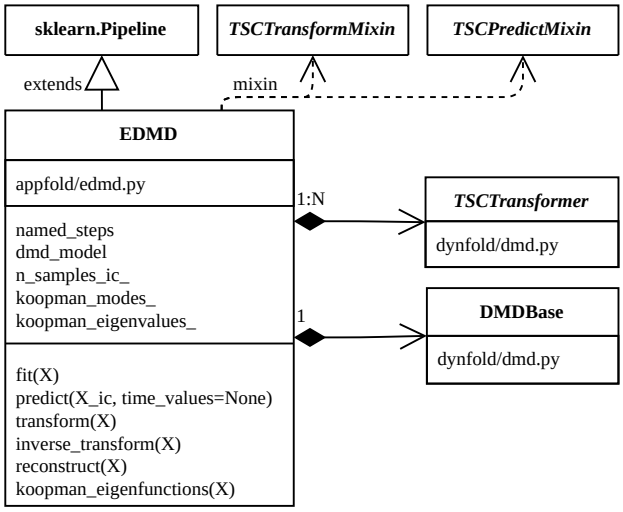


Figure 3.12: Class diagram for `EDMD` as a subclass of *scikit-learn's* `Pipeline` class. Additionally `EDMD` has two mixins that support the data transformation to an internal state representation and system identification.

The following code snippet provides examples of how to set up `EDMD` to obtain configurations reported in the literature [Arbabi and Mezić, 2017; Klus et al., 2020; Williams et al., 2015] and also the main setting of my thesis in Section 3.1:

```
# Original EDMD formulation in Williams et al. 2015
EDMD(dict_steps=[("rbf", RadialBasisFunction(...))],
     dmd_method=DMDFull())
EDMD(dict_steps=[("hermitian", HermitianBasis(...))],
     dmd_method=DMDFull())

# Hankel DMD in Arbabi et al. 2017
EDMD(dict_steps=[("timedelay", TakensEmbedding(...))],
     dmd_method=DMDEco(...))

# Approximate the Koopman generator (gEDMD) Klus et al. 2020
EDMD(dict_steps=[...], dmd_model=gDMDFull(...))

# Main setting of this thesis (Section 3.1)
EDMD(dict_steps=[("takens", TakensEmbedding(...)),
                 ("laplace", DiffusionMaps(...))],
     dmd_method=DMDFull())
```

The first argument `dict_steps` sets up the dictionary and the second argument `dmd_method` specifies the DMD variant to perform the mode decomposition. All classes are from the *dynfold* layer. Internally, the `EDMD` initializes a `Pipeline`. This is only possible because all data-driven methods align to the *scikit-learn* pipeline.

The following code listing exemplifies a data-driven workflow when using `EDMD`:

```
X: TSCDataFrame          # time series collection
X_ic: TSCDataFrame       # initial condition

# set up the model as in the previous code snippet
edmd = EDMD(...)

edmd = edmd.fit(X)              # compute Koopman triplet
X_predict = edmd.predict(X_ic) # transform and evolve system
X_dict = edmd.transform(X)      # apply dictionary on time series

# map dictionary states to original states with Koopman modes
X_orig = edmd.inverse_transform(X_dict)

# access and analyze the Koopman triplet
edmd.koopman_modes_
edmd.koopman_eigenvalues_
edmd.koopman_eigenfunctions_(X)
```

Fig. 3.13 provides a schematic illustration of the internal pipeline principle during `fit(X)` and `predict(X_ic)`. The figure includes both a code notation in *datafold* and mathematical notation, corresponding to the statements in Eq. 2.27 – 2.32.

To fit an EDMD model the measurement time series X are first passed through the dictionary pipeline to map to the intrinsic states $\mathbf{g}(X) = Z$. In the second step, the time series Z are then passed to the specified `DMDBase` class to perform the linear system identification and mode decomposition in the intrinsic states. Finally, the `EDMD` class stores the Koopman triplet $(V, \Lambda, \xi(\mathbf{x}))$ in the attributes to describe the final linear dynamical system. All of the components are valuable for system analysis and can be accessed in the `EDMD` object (see bottom of previous code snippet).

A similar procedure is followed during `predict`, where the states are first mapped to intrinsic states. However, the target map is now to a single vector $\mathbf{z}_1$ that describes the initial condition in the intrinsic state coordinates. Importantly, to perform a prediction, all methods within the dictionary pipeline must have an out-of-sample mapping to evaluate $\mathbf{g}(\mathbf{x}_1) = \mathbf{z}_1$ for $\mathbf{x}_1 \notin X$. Finally, with the Koopman modes V and eigenvalues Λ the `EDMD` model describes the state evolution of the original measurements in time, $\mathbf{x}_t$.

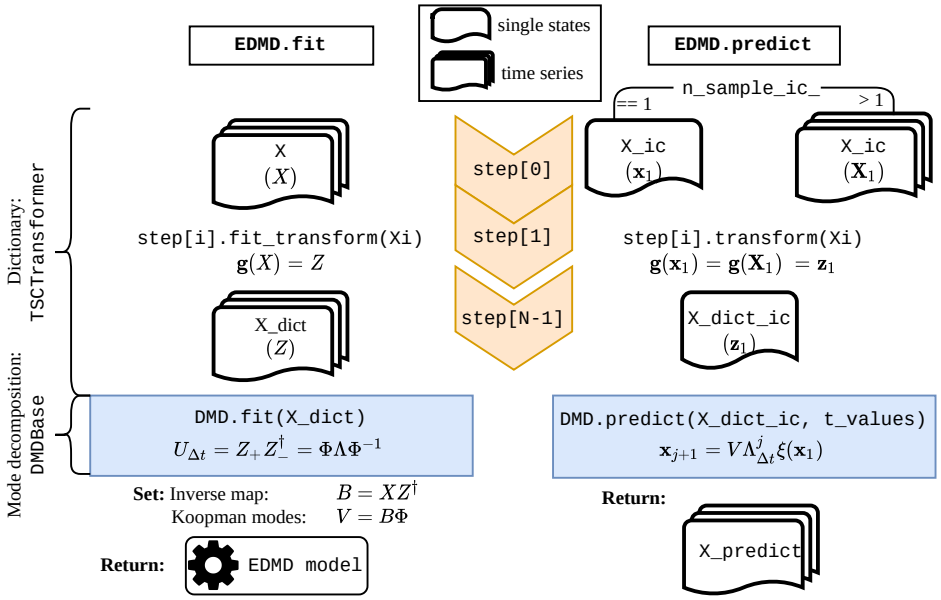


Figure 3.13: Schematic illustration of the algorithmic steps in `fit(X)` and `predict(X)` in `EDMD`. The pipeline includes N steps of sequential data transformation, corresponding to the EDMD dictionary. The final system identification is performed by a DMD variant. The mathematical statements correspond to Eq. 2.26 – 2.32.

If a EDMD dictionary $\mathbf{g}$ includes a temporal transformation method — such as the time delay embedding — the initial condition in the measurement space is required to be a time series itself. This is because the time delay embedding requires a time series of $d+1$ samples to perform the embedding to a single state, which in Fig. 3.13 is highlighted by $\mathbf{g}(X_1) = \mathbf{z}_1$. In the `EDMD` class the number of measurement states required to define an initial condition is managed in the `EDMD` attribute `n_samples_ic_`, which is automatically obtained by tracking the number of samples that are reduced during `fit`.

Fig. 3.13 includes the typical case of a single processing flow where data transformations are chained in a series. However, in a more general setting, it is also possible to split and merge separate data flows. The pipeline is then expressed as a directed graph. This is particularly interesting for heterogeneous data quantities, where each quantity has a separate dictionary. However, exploring these aspects is left to future work.

3.5.2 Parameter optimization and validation of EDMD

According to the system identification loop of Section 2.5.1 (Fig. 2.9), the `EDMD` class provides an operator-based model architecture with a linear and spectral representation of the dynamics (as per Eq. 3.8). Moreover, the model complexity can be specified via the choices of dictionary and mode decomposition in the `EDMD` class. In this section, I address the final steps within the system identification loop: (1) to find a suitable model parametrization and (2) to validate a model (as the condition in the loop). Both of these

steps are essential in a data-driven workflow to construct a model that gives strong empirical evidence to be practically useful.

Despite the importance of systematic parameter optimization and model validation, I found that these aspects are usually missing in the operator-based literature. A motivating example is given in Kutz et al. [2016c], where four different kernels are compared in a EDMD dictionary to approximate the Schrödinger equation. Since the study shows that all kernels produce different Koopman spectra (Fig. 5 in the paper), this highlights the necessity to have a structured procedure to assess the quality of each dictionary configuration and parametrization. In machine learning, the established procedures are derivative-free optimization routines and cross-validation.

The concrete set of parameters in `EDMD` depends on the actual choice of dictionary and mode decomposition. For most of the parameters, the specification is not known *a priori* because they depend on the underlying system and time series characteristics. In this section, I describe `EDMDCV` as a new class that can systematically optimize the parameters in a `EDMD` model and can be used to perform the final validation.

For both the parameter optimization and final model assessment it is important to obtain unbiased error estimates of the model. This is achieved by evaluating the model on initial conditions and their respective time series that are not part of the model fit. Therefore, the time series data is split into two separate datasets:

- The **test data** (`X_test`) are completely separated and only used for the final model validation. Evaluating the model on this data estimates the generalization error.
- The **training data** (`X`) are used for the model construction. In parameter optimization routines the training data are again split by a cross-validation scheme, where one part is used to fit the model (`X_fit`) and the other to evaluate the model's error (`X_validate`). Often a cross-validation scheme has multiple splitting configurations to better make use of the data [cf. Bishop, 2006, Fig. 1.18].

See Fig. 3.15 for an illustration of splitting data into test and training sets and two split configurations.

In accordance to *datafold's* software architecture and design decisions (see Section 3.2.2), the `EDMDCV` class reuses functionality from the previous layers and also integrates existing functionality from *scikit-learn*. Fig. 3.14 displays the class diagram of `EDMDCV`.

The main class to perform systematic time series splits as part of the parameter optimization is `TSCCrossValidationSplit`. The (abstract) class is described in Section 3.3.1.3 and located in the *pcfold* layer because it directly associates to the data structure `TSCDataFrame`. The `EDMDCV` is initialized with an arbitrary splitting strategy which can be selected according to the concrete application. To perform the actual parameter optimization, `EDMDCV` extends the base class `GridSearchCV` from *scikit-learn*. The class provides a naive optimization routine that performs an exhaustive search over a parameter grid. The candidate parameter set with the highest score is used.

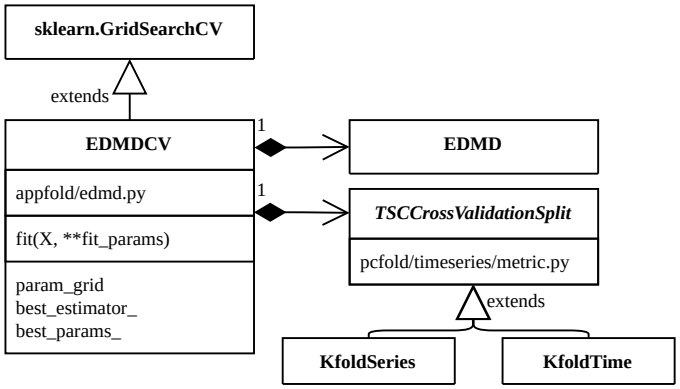


Figure 3.14: Class diagram of `EDMDCV`. The class has the `GridSearchCV` from *scikit-learn* has a base class and is a composite of an `EDMD` instance and a cross-validation strategy in `TSCCrossValidationSplit`. The optimization is an exhaustive search over a specified parameter grid, where the cross-validated scores are maximized. For an illustration of see Fig. 3.15.

The following code snippet exemplifies how `EDMDCV` is initialized and used. The three arguments of the class are (1) the `EDMD` model to be optimized, (2) the cross-validation splitting strategy and (3) the parameter grid:

```
X: TSCDataFrame        # data used for parameter optimization
X_test: TSCDataFrame   # test data for model validation

param_grid = dict(par1=[0.1, 0.2, 0.3], par2=[0.1, 0.2, 0.3])
                # EDMD specification, Section 3.5.1
edmdcv = EDMDCV(edmd=EDMD(...),
                # Cross-validation split, Section 3.3.1.3
                cv=KfoldTime(...),
                param_grid=param_grid)

# perform parameter optimization on training data
edmdcv.fit(X)

# perform model validation
gscore = edmd.best_estimator_.score(X_test)
```

The entire data-driven workflow of `EDMD` and `EDMDCV` is also visualized in Fig. 3.15. The figure highlights the interplay between `EDMD` and `EDMDCV` to cover the entire system identification loop. As already highlighted, the `EDMDCV` performs the model optimization using only the data in the training data (`X`). In the example, the parameter grid consists of two parameters with three samples each. This leads to a grid of $3^2 = 9$ parameter pairs. For each candidate pair, the cross-validation error is computed by using the splitting strategy specified in the second argument of `EDMDCV` (here `KfoldTime` from Section 3.3.1.3).

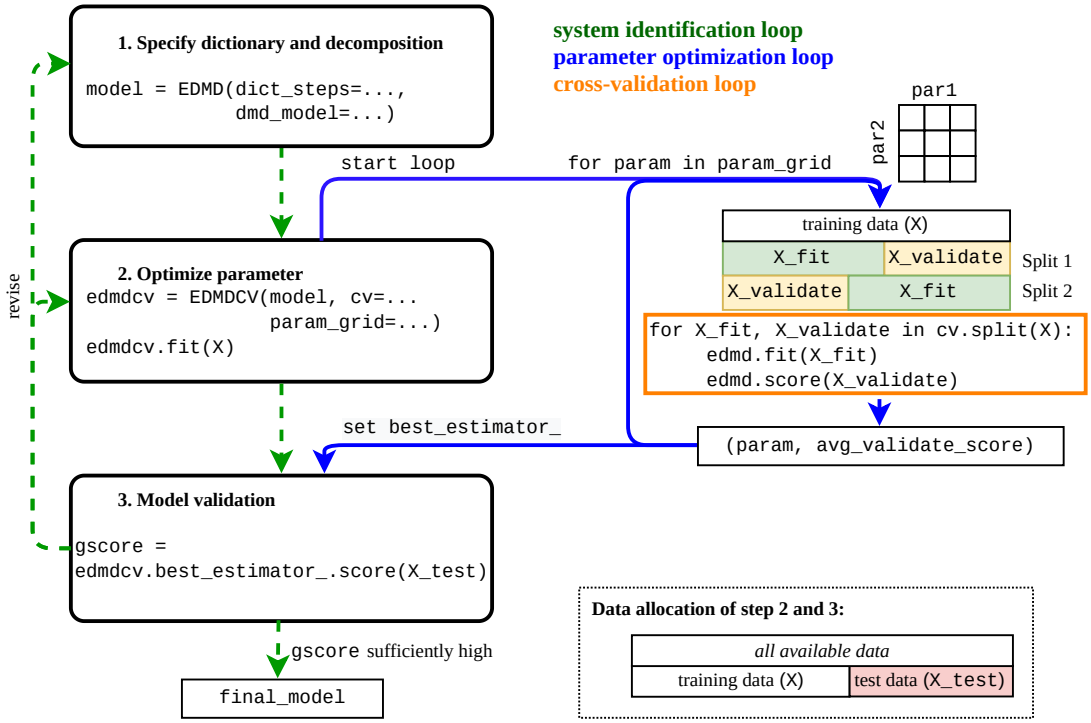


Figure 3.15: Schematic overview of the interplay between `EDMD` and `EDMDCV` within the system identification loop (corresponding to Fig. 2.9). A modeler (manually) revises the `EDMD` model specifications (step 1). The `EDMDCV` performs an automated parameter optimization on the training data by maximizing the average validation score in a cross-validation scheme (step 2). The model with the best (average) validation score (`best_estimator_`) is then again validated on separate test data (`X_test`) to estimate the generalization score as the loop condition (step 3).

The `EDMD` model with the parameter set that achieves the highest (average) score is selected as the final model (`best_estimator_` attribute in `EDMDCV`). To assess the practical relevance of the model it is necessary to validate it on separate test data. This obtains the best estimate for the generalization error because the strict separation avoids too optimistic validation results as the data are not used for the model construction process. In Fig. 3.15 this is exemplified by computing the generalization score (`gscore`) on the test data. A sufficiently high generalization score (depending on the problem) would then terminate the system identification loop. The model can then be used for its intended application. Note that the model validation can also include a qualitative analysis, where a modeler can investigate structural errors between the predicted and test time series to better guide the revision of `EDMD` and `EDMDCV`.

In practice, the generalization of `EDMD` can introduce challenges, which I describe in the following list:

Maintaining temporal order

Most of the standard splitting procedures, such as k-fold cross-validation, assume

static data that are independent and identically distributed (i.i.d.). This assumption is violated for temporally ordered time series data. Consequently, separated validation or test data may not be completely unbiased from data used to fit a model. Moreover, the data can only be separated into coherent time series (unlike random samples) to maintain the sampling rate Δt as an essential characteristic of the time series.

A common splitting strategy for time series is to only separate data at the end of a time series, to avoid blocks of missing data in the time series and dependencies between the separate data. However, Bergmeir and Benítez [2012] show that making use of cross-validation despite the "theoretical flaws" leads to more robust model selection. Fortunately, because I put emphasized the value of time series *collections* in *datafold*, splitting schemes that separate intermediate blocks of time series are possible. For example, taking an intermediate block from a single time series leads to a collection of two training and one validation time series.

Computational complexity
The exhaustive grid search performed in `EDMDCV` prohibits optimizing a large parameter grid. This is because of a combinatorial explosion, where each parameter introduces a new dimension in the search space. Overall, there are three loops involved in Fig. 3.15, which quickly add up to a large number of iterations to fit and validate a model. The time needed to fit a model is therefore a limiting factor in how thoroughly the optimization can be performed. Based on the `EDMDCV` class it is easy to extend the parameter search to sophisticated optimization strategies that try to reduce the number of (expensive) model evaluations. One method frequently used for model selection tasks is a Gaussian Process (in this context also referred to as Bayesian optimization) [Snoek et al., 2012].

Maintain same validation data for each parameter configuration
The `EDMD` class also supports temporal feature extraction methods, such as time derivatives or time delay embedding. The parameters in these methods can affect the number of states that are required for an initial condition, for example, the number of delays. Including such parameters in the parameter grid, can change the allocation in validation data. However, for a fair model comparison, the validation data should remain constant for each candidate parameter set. As a solution, it is possible to either optimize such parameters separately or include additional logic in a splitting method.

Parameter dependencies within EDMD
Because an `EDMD` model can contain multiple models in the pipeline, the parameters may depend on one another. In such cases, it can be difficult to set up a single and consistent parameter grid. An example where such dependencies frequently occur are data transformations that change the pairwise distances for a follow-up kernel method. This is the case in the main EDMD setting of this thesis in Section 3.1. The number of delays in the time delay embedding change the dimension and pairwise distances, which also affects the and therefore the optimal kernel bandwidth in the

subsequent Diffusion Maps. One solution is to optimize parameters separately via `EDMDCV`.

3.5.3 Comparing EDMD to other software projects

During my thesis, new projects have been initiated that provide general-purpose software to approximate the Koopman operator or generator. This highlights the pressing need for software to bring the operator-based methodology for system identification forward. In this section, I compare the functionality of other projects with my proposed `EDMD` class.

I only compare packages that are intended as scientific software to address general purpose problem settings. This excludes publicly available source code which is intended for proof of concept or to reproduce published results. In total there are five related software projects, which I briefly describe below. Only the first three projects are included in the more detailed comparison in Table 3.7.

PySINDy [de Silva et al., 2020, v1.4.3] — The Python package provides the Sparse Identification of Nonlinear Dynamical systems (SINDy) method [Brunton et al., 2016], which relates to the Koopman *generator*. The method therefore requires approximating the time derivative of the data. The idea of the method is to perform sparse regression to find a parsimonious set of nonlinear terms in the dictionary that best describe the dynamical system. This means that instead of linearly evolving the identified system according to the Koopman theory, ***PySINDy*** numerically integrates *nonlinear* dictionary terms with differential equation solvers. The connection between SINDy and EDMD is highlighted in Klus et al. [2020].

deeptime [Hoffmann et al., 2021, v0.2.9] — The Python package implements a variety of methods for time series analysis and modeling. This includes dimension reduction methods, clustering or Markov models. In the scope of my thesis this includes basic DMD variants, EDMD and SINDy.

DataDrivenDiffEq.jl [Martensen and Rackauckas, 2021, v0.6.6] — The package provides data-driven methods to identify dynamical systems, including basic DMD variants, EDMD and SINDy. The source code is written in the programming language Julia[19]. To the best of my knowledge, it is the only project that provides operator-based methods that are not written in Python (under the conditions highlighted above). The package therefore follows a different workflow.

PyDMD [Demo et al., 2018, v0.4.0] — The Python package provides various DMD variants and is also accessible through a wrapper in ***dynfold***, Fig. 3.11. ***PyDMD*** was initiated in 2017 and is the earliest openly available project addressing mode decomposition of time series. However, the software design of the package does not cover the "extended" part of EDMD and is therefore excluded from Table 3.7.

[19] https://julialang.org/

pykoopman [de Silva and Kaiser, 2021, v0.2.0] — The Python package provides an implementation of EDMD to either approximate the Koopman operator or generator. The software was initiated in 2020 and is maintained by the same research group as the *PySINDy* package. While the scope of *pykoopman* directly relates to EDMD, the software is still in a very early stage with only rudimentary documentation and tests; it is therefore excluded from the comparison in Table 3.7.

Table 3.7: Comparison of functionality between other packages that approximate the Koopman operator to `EDMD` in *datafold*. (Abbreviations: arbit. = arbitrary, dict. = dictionary, gen. = generator, op. = operator)

	PySINDy	***deeptime***	***DataDriven DiffEq.jl***	***datafold*** `EDMD`
initiated (year)	2019	2019	2019	2019
Koopman op./gen.	gen.	op.	op./gen.	op./gen.
expose Koopman triplet	no	yes	yes	yes
time series collection	yes	yes	no	yes
chaining methods in dict.	yes	no	no	yes
temporal transformation	no	no	no	yes
arbit. mode decomposition	yes	no	yes	yes
cross-validation	yes	no	no	yes
streaming	no	no	yes	no
control	yes	no	yes	no
Frobenius-Perron op.	no	yes	no	no
learning the dict.	no	no	no	no

All Python projects mimic the *scikit-learn* API and build on the scientific computing stack (cf. Fig. 2.5). However, because there are no established standards for a system identification API and time series data structures, the packages diverge and are not fully compatible.

In Table 3.7 only the two packages *DataDrivenDiffEq.jl* and *deeptime* provide an implementation of the EDMD and are therefore suitable for a direct comparison with my contributed `EDMD` class. As highlighted above, the package *PySINDy* has a different focus on sparse system identification of the Koopman generator.

The table highlights the distinctive features of the `EDMD` class. The modular design allows chaining methods in the dictionary pipeline. A distinctive feature is that the dictionary supports temporal feature extraction methods. Furthermore, `EDMD` supports the specification of an arbitrary mode decomposition and performing cross-validation with time series collections.

All software projects in Table 3.7 provide a flexible and arbitrary dictionary selection as an essential requirement to provide a generic framework to approximate the Koopman operator or generator. However, the packages only include “standard” function

families, such as radial basis functions or Fourier coefficients. In the next chapter, I show that with `EDMD` it is possible to integrate and chain time delay embedding and the Diffusion Maps within the `EDMD` dictionary (as promoted in Section 3.1). In all of the other packages, this would only be possible with additional implementation efforts.

On the other hand, there are also features that I have not covered in `EDMD`, which are available in other packages. This includes control, a streaming setting, and approaches to "learn the dictionary". I refer to these features in the future directions of Section 5.2.

3.6 Summary

In this chapter, I transferred the mathematical concepts of Chapter 2 to a scientific software solution: *datafold*. The software organizes these methods in a machine learning setting with the objective to analyze the geometry and perform system identification based on time series collection data. The transfer was necessary because available (partial) software solutions miss important methodological features for these tasks. While I designed *datafold* to be modular and flexible for model exploration, I also highlighted a specific operator-informed model architecture. In the first step, a spatio-temporal feature extraction is applied on the measurement data, comprising time delay embedding and Diffusion Maps. In the composed transformation this approximates the eigenfunctions of the Laplace-Beltrami operator on the state space manifold. In a second step, these new time series are used within the Extended Dynamic Mode Decomposition (EDMD) to approximate the Koopman operator for system identification. Overall, this setting is the cornerstone of my thesis to extract geometric and dynamic coordinates from time series data via the two operators.

For *datafold* I set up a sustainable software management to reproduce existing and conduct new research within the operator-informed modeling approach. In the software architecture, all methods are organized in three layers. These give a hierarchical order that provides orientation along the system identification loop and promotes algorithmic experimentation within a clearly defined data-driven modeling workflow. In the second and third layer, the methods are further organized by mixin classes that indicate the scope of a method (feature extraction and/or system identification) and augment shared code to perform the respective task. To avoid re-implementations of existing and well-tested scientific software, I based *datafold* on Python's scientific computing stack. The machine learning software *scikit-learn* is here particularly relevant because *datafold* mirrors the user interface and reuses and adapts functionality for the contributed methods if applicable.

The first layer *pcfold* includes a data structure to store time series collection data (`TSCDataFrame`). Together with associated basic functionality, the data structure is the key element to generalize system identification methods to various forms of temporal sampling schemes. The layer also includes an algorithmic contribution of basic functionality that directly operates on the data structure. With the "project-search-pullback" idea, implemented in *rdist*, the objective is to accelerate the computation of a sparse (radius-based) distance matrix.

The middle layer *dynfold* includes methods to extract or augment information in time series data to obtain a new state representation. I focused on time delay embedding and the Diffusion Maps framework as elements of the main model setting. Furthermore, *dynfold* provides basic variants of the Dynamic Mode Decomposition (DMD) to perform linear system identification.

The highest layer *appfold* provides a software solution to capture the EDMD framework. The main objective is to approximate the Koopman triplet (modes, eigenvalues and eigenfunctions) to perform nonlinear system identification. The design of the EDMD class mirrors a typical machine learning pipeline, in which multiple methods from the previous layer can be bundled in a single "meta-model". The EDMD dictionary includes one-to-many methods for feature extraction (the extended part) and the final mode decomposition is performed by one of the DMD variants. Ultimately, the class can reflect the main operator-based setting of my thesis to extract geometric and dynamical coordinates from time series collection data. In addition, *appfold* includes a class to systematically optimize the parameters contained in a EDMD model by minimizing a cross-validated error. Together the two classes capture the entire system identification loop where an EDMD model can be revised.

All of the functionality provided in *datafold* is integral to the data analysis in the next chapter. The efficient experimentation and model exploration within a data-driven modeling workflow is essential for the data applications.

4 Data analysis to extract geometry and dynamics from time series data

In this chapter, I transfer the theoretical foundation of the operator-based methods to three concrete data applications. The time series analysis is made possible through the software implementation in *datafold*, which provides a robust and well-tested environment to enable flexible model revisions within the system identification loop. The software provides the main operator-informed setting of my thesis, in which I combine the different aspects of (1) reconstructing dynamics, (2) extracting meaningful geometric coordinates and (3) performing nonlinear system identification. Within points (2) and (3) I approximate the Laplace-Beltrami and Koopman operators respectively, which describe the operator-informed setting of Section 3.1.

In Section 4.1, I analyze data from a pendulum system as an illustrative example with available equations. I first re-visit the three main methodological components separately and then combine them into a single data processing pipeline that mirrors the full operator-informed setting. I show how I can reconstruct meaningful physical variables of time series that only include partial information of the system. Furthermore, I accurately estimate the pendulum's state evolution until it reaches the equilibrium state with a linear dynamical system described by the Koopman matrix.

In Sections 4.2 and 4.3, I analyze two systems from pedestrian dynamics. This research field is confronted with partial and noisy observation data for which analytical equations are typically not available. An increased understanding of such traffic systems can improve the safety of crowd movements, which makes the approach in which the model's components are accessible particularly interesting. Both analyzed systems have fundamentally different characteristics in terms of data sources and dynamics. For the first system — simulated pedestrian movement at a bus station — I construct a coarse-grained surrogate model from the underlying microscopic simulator. The dynamics are characterized as transient. For the second system, I investigate real-world and multivariate sensor measurements of pedestrian traffic from Melbourne, Australia. For this dataset, I assume an ergodic state evolution with periodic patterns.

The last Section 4.4 includes a benchmark analysis of *rdist* as an additional contribution to my thesis to efficiently compute a sparse distance matrix (following from Section 3.3.2.2). I compare my implementation against state-of-the-art implementations in high-performance libraries on both simulated and real-world datasets.

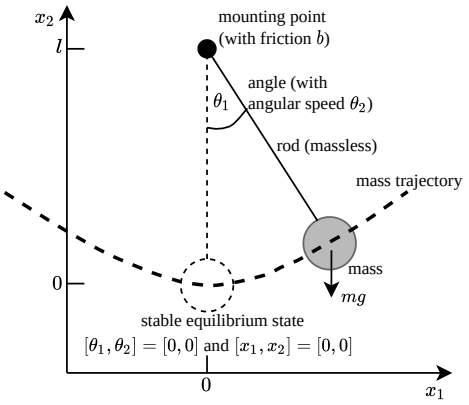
All data and source code that I use in this chapter is available in the Supplementary Material.

4.1 Pendulum: Extracting geometric and dynamic coordinates

The pendulum is a classic example of a physical dynamical system. I use the pendulum to illustrate the functionality of *datafold* and to showcase the capabilities of the main components of the operator-informed setting for a system with known equations. After I describe the system and the sampled time series data in Section 4.1.1, I go through the main steps of **(1)** time delay embedding (Section 4.1.2) **(2)** geometric analysis with DMAP to approximate the eigenfunctions of the Laplace-Beltrami operator (Section 4.1.3) and **(3)** construct a Koopman operator-based model with EDMD to identify the pendulum system (Section 4.1.4).

4.1.1 Description of the dynamical system and data collection

The pendulum can be described by a first-order ordinary differential equation (ODE) system. By describing the dynamical system in angular coordinates in Eq. 4.1, the ODE has two state quantities: the pendulum's position θ_1 and the angular speed θ_2). This system is in a closed-form and parsimonious representation. The system equations can be derived from a second-order ODE (i.e. in terms of a second derivative of the position, $\ddot{\theta}$) from Newton's 2nd law of motion. The system example is taken from the web page [Hubbs, 2020], which provides more detailed explanations of the pendulum and includes the Python code for solving the ODE numerically.



$$\begin{aligned}\dot{\theta}_1 &= \theta_2 \\ \dot{\theta}_2 &= -\frac{b}{m}\theta_2 - \frac{g}{l}\sin(\theta_1)\,, \end{aligned} \tag{4.1}$$

Figure 4.1: Schematic illustration of the pendulum system with physical parameters (b = friction, g =gravitation, l = length of rod, m = mass). The pendulum can be observed through the angular (θ_1, θ_2) and Cartesian (x_1, x_2) coordinate system. The graphic is adapted from Hubbs [2020].

The system parameters specify the actual physical properties of the pendulum and are visualized in Fig. 4.1: A mass with $m = 1\mathrm{kg}$ is attached to a rod of length $l = 1\mathrm{m}$, with a positive friction $b = 0.45$ at the mounting point. The gravity acting on the mass is $g = 9.81\mathrm{m/s^2}$. To simplify the general setting, I do not consider the case in which the pendulum flips over.

While the analytic differential form of the pendulum is available, I treat the system as a black box for which only time series from a given initial condition can be generated. To have a manageable dataset for the illustrative example, I only sample the system at three initial conditions with resulting time series:

$$[\theta_1, \theta_2]_{j=1}^{(i=1)} = \left[\frac{\pi}{3}, -4\right] \qquad \Rightarrow [\theta_1^{(1)}, \theta_2^{(1)}, \ldots, \theta_{500}^{(1)}] \tag{4.2}$$

$$[\theta_1, \theta_2]_{j=1}^{(i=2)} = \left[-\frac{3\pi}{4}, 2\right] \qquad \Rightarrow [\theta_1^{(2)}, \theta_2^{(2)}, \ldots, \theta_{500}^{(2)}] \tag{4.3}$$

$$[\theta_1, \theta_2]_{j=1}^{(i=3)} = \left[\frac{\pi}{2}, -2\right] \qquad \Rightarrow [\theta_1^{(3)}, \theta_2^{(3)}, \ldots, \theta_{500}^{(3)}], \tag{4.4}$$

where j is the time index and i the time series. According to the specified physical properties, all time series end up in the stable equilibrium state at $[\theta_1, \theta_2] = [0, 0]$ (the system has a second (unstable) equilibrium point at $[\theta_1, \theta_2] = [\pi, 0]$).

Numerically integrating the ODE system in Eq. 4.1 with the three initial conditions results in a time series collection of three time series in angular coordinates. For the numerical integration I use the standard Runge-Kutta 45 from the ***SciPy*** package [Virtanen et al., 2020]. For each time series from the initial conditions I sample 500 measurements between $t_1 = 0$ and $t_{500} = 8\pi$, resulting in a time increment of $\Delta t \approx 0.05$.

An alternative observation to the angular state quantities is a Cartesian coordinate system. Here the pendulum is tracked in the coordinates (x_1, x_2), which can be obtained by mapping the angular states:

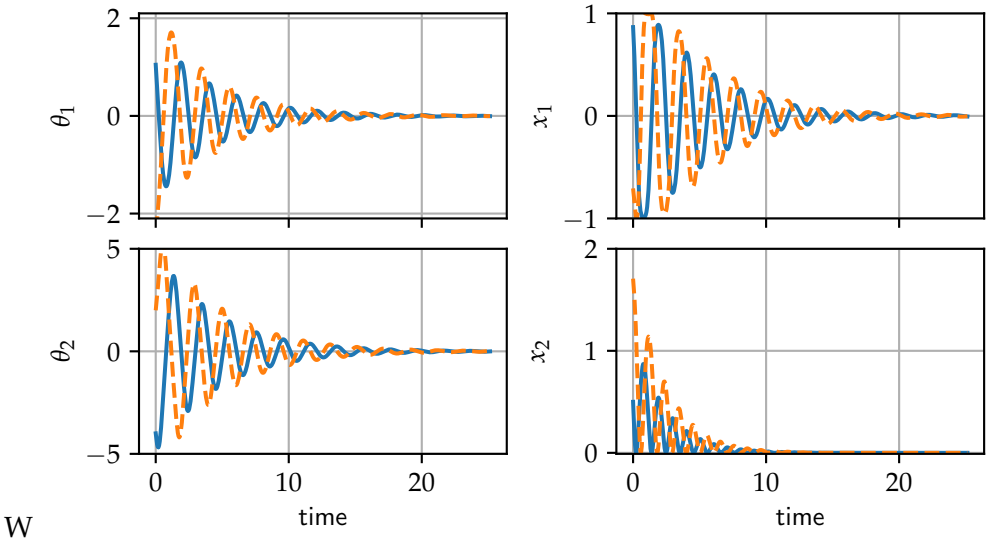
$$\mathbf{x}_j^{(i)} = [x_{1,j}^{(i)}, x_{2,j}^{(i)}]^T = \left[l \cdot \cos\left(\theta_{1,j}^{(i)} - \frac{\pi}{2}\right), l\left(1 + \sin\left(\theta_{1,j}^{(i)} - \frac{\pi}{2}\right)\right)\right]^T, \tag{4.5}$$

where $j = 1, \ldots, 500$ is the time index and $i = \{1, 2, 3\}$ the time series index, corresponding to the initial condition. The pendulum's position is relative to the mounting point at $[x_1, x_2] = [0, l]$. See Fig. 4.1 for a visualization of the two reference systems. Note that the time series can be stored and managed in ***datafold's*** main data structure `TSCDataFrame` from Section 3.3.1.1.

Both the angular and Cartesian coordinate systems describe a two-dimensional state. However, Eq. 4.5 represents a projection from a well-defined system state to the Cartesian coordinates in which dynamical information is lost. Unlike the angular states which are obtained from a closed-form ODE system, the Cartesian coordinates are no longer well-defined. With only a single measurement state (x_1, x_2), it is impossible to determine the next future state without further temporal context: The pendulum may swing in either direction or be at rest.

In the pendulum example, I assume that we have only access to time series in Cartesian coordinates. This could be a result of a selection of measurement quantities or we may be limited in the observation modality. For example, we could only have video material of the pendulum from which we can track the Cartesian coordinates.

Furthermore, I only use the first two time series ($i = 1, 2$) to build a model. Fig. 4.2 displays the solution of the both resulting time series in angular (left) and Cartesian coordinates (right). The third time series for the comparison with out-of-sample data.



W

Figure 4.2: The orange (dashed) time series are generated with the first initial condition and the blue (solid) time series with the second in Eq. 4.2 – 4.3. The left column displays the hidden system states in angular coordinates (θ_1, θ_2) and the right column shows the measurement states in Cartesian coordinates (x_1, x_2) from Eq. 4.5.

In the following sections, I use the methods provided by *datafold*: First I use time delay embedding to reconstruct the dynamics in the Cartesian coordinates (Section 4.1.2). Based on the reconstructed states I extract geometrically meaningful coordinates (Section 4.1.3). Finally, I use the coordinates to perform system identification with DMD and EDMD to predict the trajectory of the pendulum (Section 4.1.4).

4.1.2 Reconstructing partial observations with time delay embedding

As highlighted in the previous section, the time series describing the pendulum's trajectory in a Cartesian reference system are not well-defined. In this section, I apply time delay embedding to reconstruct the dynamics by augmenting each state (x_1, x_2) with prior samples. For this I use the class `TakensEmbedding` from Section 3.4.2. The geometric picture of the time delay embedding is described in Section 2.4.3.

Takens theorem provides general guidance of how many delays are required to reconstruct the state space: If the hidden system is on a m-dimensional smooth attractor

manifold, then an embedding with $d \geq 2m + 1$ delays is one-to-one (diffeomorphic) with the attractor [Deyle and Sugihara, 2011; Takens, 1981]. In the angular coordinate system of Eq. 4.1 the pendulum has an intrinsic state space dimension of $m = 2$ (the angular coordinates). However, because of the positive friction in the pendulum in Eq. 4.1, most states are in a *transient* regime of the state space and converge towards a "point attractor" as the steady state. This means that the assumptions of the theorem are not fulfilled for the time series data. Nevertheless, I show that the methodology is able to reconstruct the dynamics and state space geometry in this setting.

To highlight the "reconstruction effect" of the time delay embedding on the Cartesian time series, I set up a linear dynamical system on different number of delays (d):

$$\mathbf{g}_{\text{td}}(\mathbf{x}_j; d, \kappa) = [\mathbf{x}_j; e^{-\kappa}\mathbf{x}_{j-1}; e^{-2\kappa}\mathbf{x}_{j-2}; \ldots, e^{-d\kappa}\mathbf{x}_{j-d}] = \mathbf{y}_j \in \mathbb{R}^{2\cdot(d+1)}. \tag{4.6}$$

$$A\mathbf{y}_j^{(i)} \approx \mathbf{x}_{j+1}^{(i)} \tag{4.7}$$

$$\mathbf{r}^2 = \left\| A\mathbf{y}_j^{(i)} - \mathbf{x}_{j+1}^{(i)} \right\|_2^2. \tag{4.8}$$

For the analysis here the weighting factor is considered $\kappa = 0$. The system matrix A is obtained in a least squares sense to map a delayed state $\mathbf{y}_j$ to its future state $\mathbf{x}_{j+1}$ (see also the code snippet below). To measure the degree of how well the states are reconstructed I compute the squared residuals for both coordinates in Eq. 4.8, $\mathbf{r}^2 = (r_1^2, r_2^2)$. In the setting I increase the number of delays from $d = 0$ (no reconstruction) to $d = 10$.

The following code snippet highlights how the above setting can be achieved in only three statements by using ***datafold*** and the compatibility to ***NumPy*** [Harris et al., 2020]:

```
X_cart: TSCDataFrame  # time series in Cartesian coordinates

# Perform time delay embedding on pendulum data X
# for d = 0, ..., 10 (see also Sections 2.4.3 and 3.4.1)
X_d = TakensEmbedding(d=d).fit_transform(X_cart)

# Compute shift matrices
# using accessor of TSCDataFrame (Section 3.3.1.2)
Y, X = X_d.tsc.shift_matrices(orientation="rows")

# Only map to original Cartesian states
X = X[:, [0, 1]]

# Fit linear dynamical system in least squares sense with NumPy
A, sqresidues, _, _ = np.linalg.lstsq(Y, X)
```

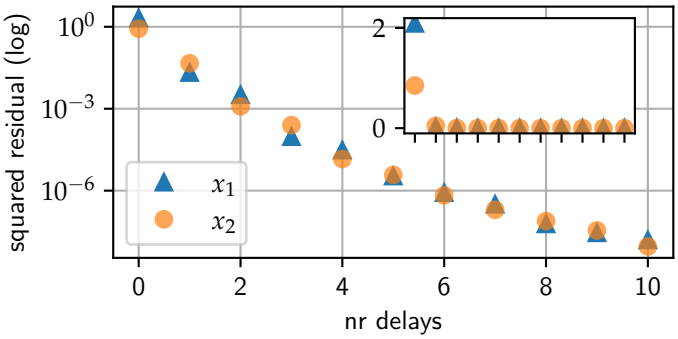


Figure 4.3: The squared residual (per coordinate) from Eq. 4.8 plotted against the number of delays (d) for the two Cartesian coordinates. The residual is plotted on a logarithmic scale. The small subplot shows a linear scale.

Fig. 4.3 plots the squared residual values obtained for both Cartesian coordinates for all number of delays $d = 0, \ldots, 10$. The residual values in relation to the pendulum layout (with a rod of length $l = 1$) are catastrophic if no time delay embedding is performed; $r_1^2 \approx 2.1$ and $r_2^2 \approx 0.9$. Providing context of a single delay ($d = 1$) already yields a reasonable mapping with residuals in an order of $\mathcal{O}(10^{-2})$. Further increasing the number of delays monotonically decreases the residuals and therefore linearizes the dynamics. This showcases why time delay embedding has become an important component in the context of the Koopman operator theory, as highlighted in Section 2.4.4.1 or in Arbabi and Mezić [2017] and Giannakis [2019]. However, including a large number of delays can become impractical in forecasting settings because a time series with $d + 1$ states is required to perform the embedding, which may not always be available.

4.1.3 Uncovering hidden state space geometry

As highlighted in Fig. 4.3, augmenting the original measurement states with prior samples reconstructs the dynamics in the delayed states. This introduces a new geometry on which the dynamics are well-defined (cf. Fig. 2.8 on page 48 and Berry et al. [2013]). However, a side effect of the time delay embedding is that the state quantities become highly correlated because of the temporal neighborhood relation in the time series data. In this section, I continue to extract the leading geometric coordinates from the delayed states to obtain a *latent* state representation of the system. For this I use Diffusion Maps (DMAP) as a manifold learning method (implementation in `DiffusionMaps`, Section 3.4.3). Together the two operations correspond to a composed data transformation:

$$\begin{aligned} \mathbf{g}(\mathbf{x}) &= [\mathbf{g}_{\text{dmap}} \circ \mathbf{g}_{\text{td}}](\mathbf{x}) \\ &= [\psi_1(\mathbf{y}); \psi_2(\mathbf{y}); \ldots; \psi_P(\mathbf{y})] = \mathbf{z} \in \mathbb{R}^P, \end{aligned} \tag{4.9}$$

which is detailed in Section 2.4.4. The eigenfunctions ψ_p are the DMAP coordinates (evaluated on the delayed states) and approximate the eigenfunctions of the Laplace-Beltrami operator (Eq. 2.44). To obtain a parsimonious set of latent coordinates it is required to identify "geometrically dependent" eigenfunctions and only select coordinates that are aligned in different directions on the manifold. The overall data processing workflow is schematically depicted in Fig. 4.4.

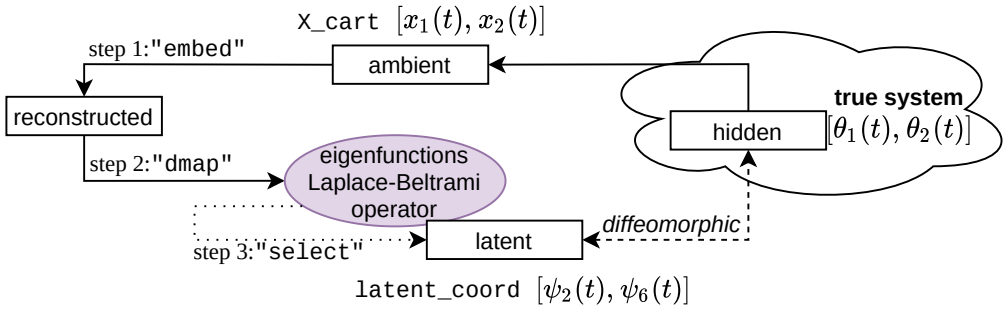


Figure 4.4: A schematic description of the data processing to extract latent coordinates that relate to the hidden state space geometry. Note that $\psi(t) := \psi(\mathbf{g}_{\text{td}}(\mathbf{x}(t)))$.

In the software treatment, the three steps `embed`, `dmap` and `select` as shown in Fig. 4.4 can be integrated in a `Pipeline` class from *scikit-learn*. The pipeline class sequentially applies the data transformations in the argument `steps`. This highlights that the *datafold* API and also the time series collection data structure remain compatible with *scikit-learn*.

```
X_cart: TSCDataFrame     # time series in Cartesian coordinates
X_cart_oos: TSCDataFrame # out-of-sample time series

# wrap three processing steps in a meta-model
pipeline = sklearn.Pipeline(
    steps=[
        ("embed", TSCTakensEmbedding(delays=3, lag=0, kappa=0.2)),
        ("dmap",
           DiffusionMaps(kernel=GaussianKernel(epsilon=3.0),
                         n_eigenpairs=6, alpha=1.0)),
        ("select", LocalRegressionSelection(intrinsic_dim=2)),
    ]
)

latent_coord     = pipeline.fit_transform(X_cart)
latent_coord_oos = pipeline.transform(X_cart_oos)
```

The first two lines of the code describe the three Cartesian time series generated from the initial conditions Eq. 4.2 – 4.4, where the first two are contained in `X_cart` and the third `X_cart_oos` is used for out-of-sample time series.

For the time delay embedding (the first data transformation) I use three delays $d = 3$ and set the weighing factor to $\kappa = 0.2$. The latter parameter is result of a cross-validation with EDMD, which I perform in the Section 4.1.4 below. During this first step, the original two-dimensional Cartesian states are embedded to a new state dimension $2 \cdot (d + 1) = 8$. The second step then computes the six leading DMAP coordinates $\{\psi_p\}_{p=1}^{P=6}$ (specified with `n_eigenpairs=6`). By setting $\alpha = 1$ the non-uniform sampling density in the point cloud is normalized, which increases the convergence guarantees to the Laplace-Beltrami operator [Coifman and Lafon, 2006b]. For the geometric prior, I use a standard Gaussian kernel with a bandwidth $\varepsilon = 3$. In the last step of the pipeline, `LocalRegressionSelection` performs an automatic selection of two geometrically informative coordinates in the six DMAP coordinates. The method is implemented in the ***dynfold*** layer and described by Dsilva et al. [2018]. The selection is performed with a correlation analysis in a local linear regression, where coordinate sets are more likely to describe independent geometrical directions on the manifold if they are highly de-correlated. By setting the target dimension to two (`intrinsic_dim=2`), I use knowledge of the hidden system. In practice the target dimension is typically not known. However, there are methods to estimate the manifold dimension [Strange and Zwiggelaar, 2014].

Overall, the pipeline describes a data transformation from two-dimensional Cartesian states in the input to two-dimensional latent states in the output. The final mapping of the time series is, $\mathbf{x}_j^{(i)} \rightarrow [\psi_2(\mathbf{x}_j^{(i)}), \psi_6(\mathbf{x}_j^{(i)})]$.

Fig. 4.5 compares different DMAP candidate pairs to justify that the final selection (ψ_2, ψ_6) by the local regression is indeed well-suited. The eigenvector ψ_1 is omitted from the start, because it is a constant coordinate — a result of the row-stochastic matrix within DMAP. The first non-trivial eigenvector ψ_2 is considered as the coordinate that aligns to the most dominant direction on the manifold. To identify the second coordinate I combine ψ_2 with the other four eigenvectors ψ_3 to ψ_6. The first row of Fig. 4.5 shows that the coordinate pairs only describe one-dimensional geometries, which can be identified as "repeated eigendirections" [Dsilva et al., 2018] (Section 2.4.2). In contrast, the final pair (ψ_2, ψ_6) in the second row (left) exhibit highly de-correlated points and capture a two dimensional geometry. As expected, this coordinate pair is also the final selection of `LocalRegressionSelection`.

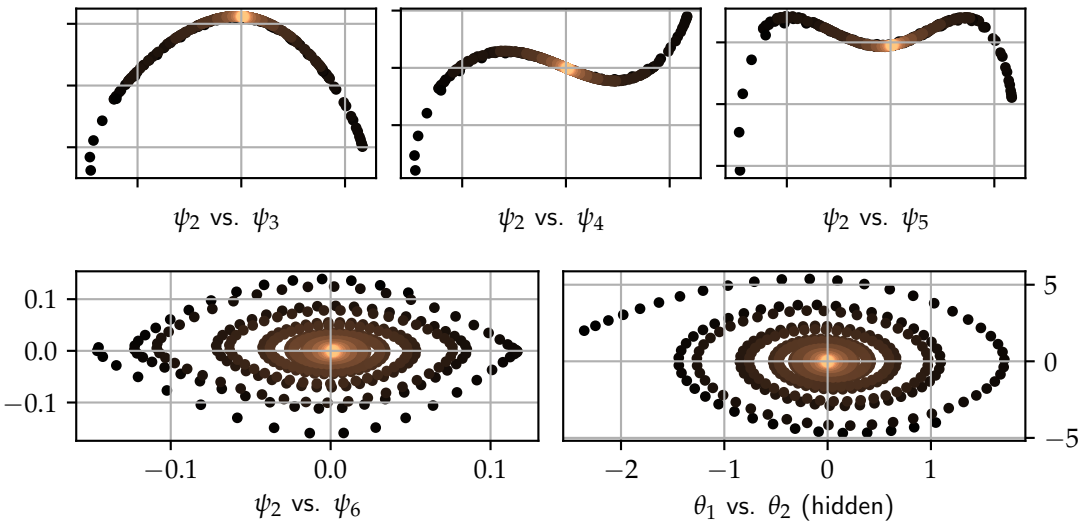


Figure 4.5: Top row: Spatial projections onto pairs of DMAP coordinates. Bottom row left: Decorrelated point cloud as the final selection of the latent state representation. Bottom row right: Original point cloud of angular coordinates. All plots: The color-code corresponds to the time value of a sample in the time series, where lighter points have a higher time value.

Comparing the final coordinate pair with the original angular time series (as samples of the hidden state space) we see that the two point clouds in the second row are on different scales but otherwise resemble the same shape. In a visual mapping ψ_2 corresponds to the angle θ_1 and ψ_6 to the angular speed θ_2. It is valid to assume that an invertible and differential one-to-one mapping — a diffeomorphism — exists between the two elliptic-shaped point clouds (as representatives of the manifolds).

So far I have performed spatial projections of the Cartesian states to a reconstructed and latent state. However, throughout the data processing pipeline, the temporal context of the time series data is maintained, making it possible to plot the latent coordinates (ψ_2, ψ_6) over time. In Fig. 4.6 (first row) I plot the out-of-sample time series (`X_cart_oos`). The `DiffusionMaps` class uses the Nyström extension to perform the map for samples that are not included in the original dataset.

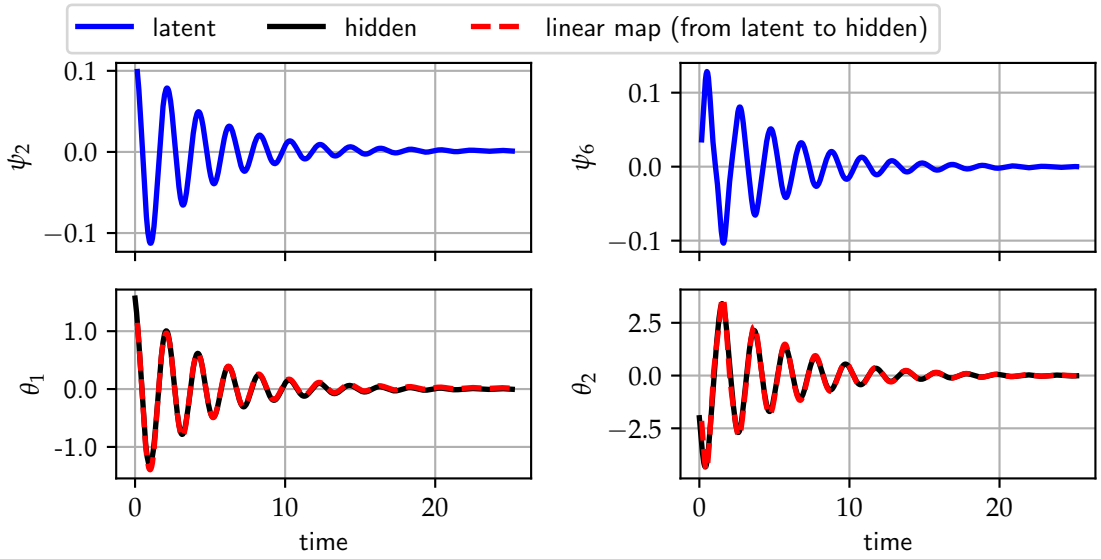


Figure 4.6: First row: Latent coordinates of the out-of-sample time series, corresponding to `latent_coord_oos` in the code snippet. Second row: Original time series in angular coordinates (position and speed) and overlaid with the time series obtained from the linear map in Eq. 4.10.

The time series in Fig. 4.6 show that the latent states $[\psi_2, \psi_6]$ resemble the hidden time series $[\theta_1, \theta_2]$ in the second row. It is apparent that the harmonic functions (describing the pendulum swings) have a similar decay over time and that the crests (and troughs) match in their positions.

To showcase a diffeomorphic relation between the two coordinate systems, I solve for a linear map

$$M \begin{bmatrix} \psi_2 \\ \psi_6 \end{bmatrix}_j^{(2)} \approx \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}_j^{(2)}, \tag{4.10}$$

that projects the latent to the (hidden) angular coordinates. Fig. 4.6 displays the resulting time series from the map (the second row, dashed). The close overlay suggests that the diffeomorphic equivalence between the latent and hidden coordinate systems also holds for the out-of-sample time series.

The reconstruction of the hidden angular state space in its parsimonious coordinates is astonishing because the Cartesian observations on which the model is built are only partially observed and do only indirectly have a notion of angle and angular speed (the inverse function of Eq. 4.5 does not exist). By projecting the time-delayed Cartesian coordinates on the eigenfunctions of the Laplace-Beltrami operator, I was able to recover physically meaningful coordinates. In usual data-driven settings, the hidden state space is unknown and such direct comparisons cannot be made. The DMAP coordinates are therefore usually treated as dimensionless latent variables that describe a

"surrogate geometry" which is believed to connect to the hidden state space. The approach contributes to the general trend in machine learning to also be able to discover "hidden physics" in complex and high-dimensional problems [Brunton et al., 2016]. In the next section I use the DMAP functions $\{\psi_p(.)\}_{p=1}^{P}$ as a basis to perform system identification within the Koopman operator framework.

4.1.4 System identification with mode decomposition

In the previous section, I focused on a geometrically motivated spatio-temporal feature extraction of Cartesian time series to obtain a more suitable state representation. In this section, I continue to identify the dynamics of the pendulum system by making use of the new state representation. This means the perspective switches from an unsupervised to a supervised learning problem, where I aim to approximate the flow of the pendulum, with system identification based on mode decomposition.

Dynamic Mode Decomposition
I first showcase a system identification with a standard DMD method. However, here I use the well-defined angular time series (previously treated as hidden). This is because the Cartesian time series have ill-defined dynamics and without augmenting further temporal context the system identification fails. For the DMD this is well-exemplified with the "standing sine wave" in Tu et al. [2014]. The intention of the following analysis is to highlight that even when the exact angular system states are available, the standard DMD has limited capacities. This further motivates the dictionary choice when utilizing EDMD below.

The following code I use the `DMDFull` class from Section 3.4.4 to identify the pendulum system:

```
X: TSCDataFrame        # time series in angular coordinates
X_oos: TSCDataFrame    # out-of-sample time series

# perform decomposition on pendulum data
dmd = DMDFull().fit(X)

# reconstruct time series in X with DMD model
X_reconstruct = dmd.reconstruct(X_oos)

# compute difference time series
X_diff = X_oos - X_reconstruct
```

For the time series data in `X` only the first two generated time series for the pendulum are used. Note that already at this stage a mode decomposition would not be possible with the *PyDMD* package because it does not support time series collections. In the DMD model the dynamics of the angular time series captured in a linear dynamical system with a matrix of size $U_{\Delta t} \in \mathbb{R}^{[2\times 2]}$ (see Eq. 3.6 – 3.7). While this setting is ideal

to visualize and interpret the local dynamics of the time series, the size of the matrix already indicates that it is not able to capture the nonlinear angular state evolution of the pendulum.

Fig. 4.7 displays both the true (left) and the identified (right) vector field. Both vector fields are generated by simulating short time series conditions in a structured grid of $\theta_1 = (-\pi/2, \ldots, \pi/2)$ and $\theta_2 = (-4, \ldots, 4)$ (ten sampling points each). Additionally, each vector field contains the out-of-sample time series.

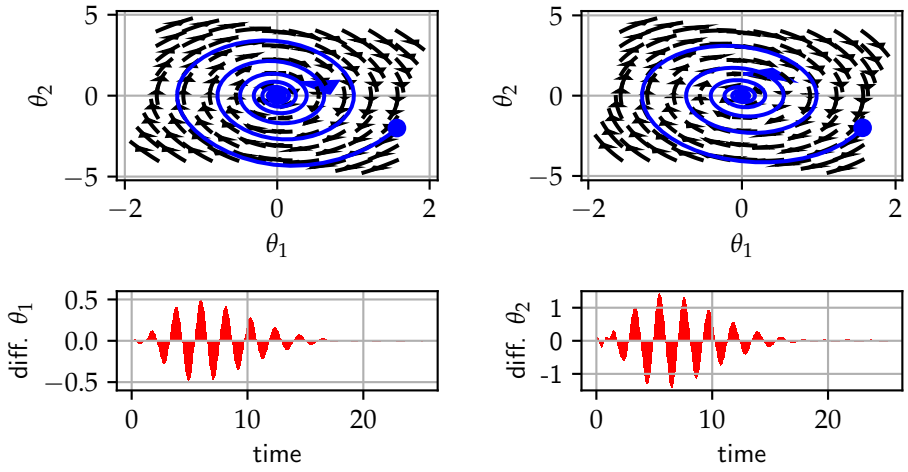


Figure 4.7: First row: The vector field on the left is from the original pendulum system in Eq. 4.1 with a trajectory from Eq. 4.4. On the right is the vector field of the identified DMD system and respective trajectory prediction with same initial condition. Second row: The difference time series between the original and identified trajectory for both angular coordinates.

When visually comparing the true and identified vector field (top row of Fig. 4.7) the DMD model appears to match the true underlying system well and the out-of-sample trajectory also seems to approximate the original system. However, when plotting the difference time series (bottom row of Fig. 4.7), we see that the prediction of the pendulum becomes out-of-sync with the original system. After a short time, the difference time series starts to oscillate to differences up to half of the rod's length $l = 1$. Both the true and predicted trajectories include a blue arrow after $t = 4$. The mismatch results from the fact that the DMD model assumes linear dynamics. It is therefore incapable to capture the nonlinear state evolution, even when the states are "perfect" in that they are directly generated from an ODE. Nevertheless, the DMD can be a useful analytic tool for the analysis of dynamics over a short-time horizon.

Extended Dynamic Mode Decomposition

In the final step, I bundle all of the previous analysis and methods into a single EDMD model. The goal is now to perform nonlinear system identification on the Cartesian data, by making use of the full operator-informed setting. In particular, I use the composed data transformation of Eq. 4.9 (the eigenfunctions of the Laplace-Beltrami op-

erator) as a geometrically-aligned basis in the EDMD dictionary to approximate the Koopman operator.

The following code includes the main interaction points with *datafold* to set up the model and optimize its parameters by minimizing the cross-validation error:

```
X: TSCDataFrame          # all three time series of the pendulum
X_test: TSCDataFrame     # solution time series for test
X_ic: TSCDataFrame       # initial condition of X_test

# dedicated class to split pendulum data
# time series i={1,2} -> build the model
#             i=3     -> compute validation error
class PendulumSplit(TSCCrossValidationSplit):
    def split(self, X, y=None, groups=None):
        index_ids = X.index.codes[0]
        fit_indices = np.where(np.isin(index_ids, [0, 1]))[0]
        validate_indices = np.where(index_ids == 2)[0]
        yield fit_indices, validate_indices

# set up EDMD with dictionary and mode decomposition
edmd = EDMD(
    dict_steps=[ ("delay", TakensEmbedding(delays=3)),
                 ("dmap", DiffusionMaps()),
    ],
    dmd_model=DMDFull(),
    sort_koopman_triplets=True,
)

# specify a grid of three parameters, with three samples each
edmdcv = EDMDCV(
  estimator=edmd,
  param_grid={  "delay__kappa": [0, 0.75, 1.5],
                "dmap__n_eigenpairs": [30, 40, 50],
                "dmap__kernel":
      [GaussianKernel(epsilon=eps) for eps in [1, 3, 5]],
  },
  # splitting strategy for parameter optimization
  cv=PendulumSplit(),
  # perform optimization in parallel
  n_jobs=-1,
)

# Figure 4.8
edmdcv.fit(X)
```

```
edmd_best = edmdcv.best_estimator_

# Figure 4.9
X_pred = edmd.predict(X_ic, time_values=np.arange(3*dt, 11, dt))

# Figure 4.10
edmd_best.koopman_modes_
edmd_best.koopman_eigenvalues_
edmd_best.koopman_eigenfunctions_(X_test)
```

All three sampled Cartesian time series of the pendulum are contained in `X` and represent the available data. In the code snippet, the `EDMD` model describes the main setting of my thesis to extract geometric and dynamic coordinates from time series data (cf. Section 3.1 and Fig. 3.1). The `EDMD` dictionary as in Eq. 4.9 for the previous geometrical state space analysis. This means that the latent coordinates $[\psi_2, \psi_6]$ that connect to the hidden state space geometry remains. However, for the EDMD there is no coordinate selection. Instead, all DMAP coordinates $\{\psi_p\}_{p=1}^{P}$ (sorted by their corresponding eigenvalue in DMAP) are used within the dictionary after a truncation P (`n_eigenpairs`).

For a successful system identification with EDMD, where multiple models interact, it is important to adjust the parameters involved. Particularly for the specified function basis that I utilize, Berry et al. [2013] (p. 28) highlights that

"[...] careful adjustment of the time-delay embedding (via our weighting scheme) and the diffusion map algorithm (via careful selection of parameters) is necessary for these techniques to work together optimally."

Without prior knowledge of well-suited parameter choices, it is necessary to systematically sample the parameter space by optimizing the model error on the validation set. In ***datafold*** this is achieved with the `EDMDCV` class. As highlighted in Section 3.5.2 a main component for the parameter optimization is a splitting strategy to optimize against (unbiased) validation errors. The above code includes a dedicated strategy in `PendulumSplit`: The first two time series ($i = \{1, 2\}$) are used to fit the model and the third ($i = 3$) to compute the validation error.

The last component of performing parameter optimization is to specify a parameter grid. In the above code snippet this includes the delay weighing (`kappa`), the number of eigenvectors (`n_eigenpairs`) and the bandwidth of a Gaussian kernel (`epsilon`). Note that syntax `name__parameter` is inherited from the base class `Pipeline` of `EDMD` which allows internal model attributes to be accessed in a single point of access. For each parameter, I sample three values, which results in $3^3 = 27$ models to fit and validate. All other parameters that are not contained in the grid remain as specified in the base `EDMD`. The model with the lowest validation error is then selected as the final model. For a simpler analysis, I again use the validation time series and its initial condition in `X_test` and `X_ic`. In practice and according to Fig. 3.15 it is better to

use test data that is completely separated from the parameter optimization to obtain an unbiased estimate of the model's predictive performance.

Fig. 4.8 plots the root mean squared error (RMSE) around the optimal parameter setting, highlighted with a red marker. The middle graph shows that the steepest increase in error is caused by decreasing the number of eigenpairs in `DiffusionMaps`, which corresponds to the number of observables in the `EDMD` dictionary. Since the lowest error is at the edge (`n_eigenpairs=50`), this suggests to increase the eigenpairs even more. However, because the DMAP eigenvalues converge towards zero, the risk of numerical instabilities increases for the Nyströem out-of-sample mapping (see Eq. 2.43 on page 45). In the final setting, the smallest eigenvalue is in the order of $\omega_{50} = \mathcal{O}(10^{-13})$. The other two parameters `kappa` and `epsilon` have the lowest error at the middle value, presumably near a minimum of a convex error relation. According to the optimization the final parameter choice is `kappa=0.75`, `n_eigenpairs=50`, `epsilon=3`.

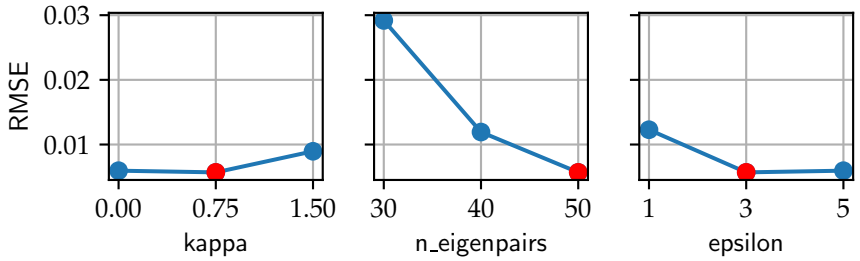


Figure 4.8: The three optimized parameters in the `EDMD` model. The red sample highlights the best parameter combination in the grid. The other two samples highlight the variation of each parameter (per plot) around the optimum in the three-dimensional grid.

Another effective parameter in the EDMD model is the `delays` in the time delay embedding. I intentionally left out the parameter from the optimization, because it changes the state dimension within the processing pipeline. Therefore, the pairwise distances for the kernel in DMAP change and there is no suitable parameter grid: a different set of `epsilon` parameters need to be sampled for each number of delays. It is thus best to optimize the parameter separately. Here I make use of the previous analysis performed of Section 4.1.2 and select `delays=3`.

Fig. 4.9 compares the best `EDMD` model against the true pendulum system, by predicting the out-of-sample time series (`X_test`). In mathematical terms the Cartesian pendulum coordinates $\hat{\mathbf{x}}$ are predicted according to obtained Koopman triplet, consisting of the modes $\mathbf{v}_p \in \mathbb{C}^2$, the eigenvalues $\lambda_p \in \mathbb{C}$ and the eigenfunctions $\xi_p(\mathbf{z}) : \mathbb{R}^{50} \to \mathbb{C}$. All of the components are computed during the EDMD model fit (for derivation see Eq. 2.26 - 2.32):

$$\hat{\mathbf{x}}_{j+1} = \sum_{p=1}^{P=50} \mathbf{v}_p \lambda_p^j \xi_p(\mathbf{z}_1). \tag{4.11}$$

The initial condition $\mathbf{z}_1$ is obtained from the time series `X_ic` (in the above code), which requires the current state and three prior states in order to perform the time delay embedding; $\mathbf{g}_{\text{td}}(\mathbf{x}_1) = \mathbf{y}_1$ in Eq. 4.6. In the plots of Fig. 4.9 this is highlighted by the fact that the prediction only starts after $3 \cdot \Delta t$ time steps. Too many delays can therefore limit the usefulness of a predictive model.

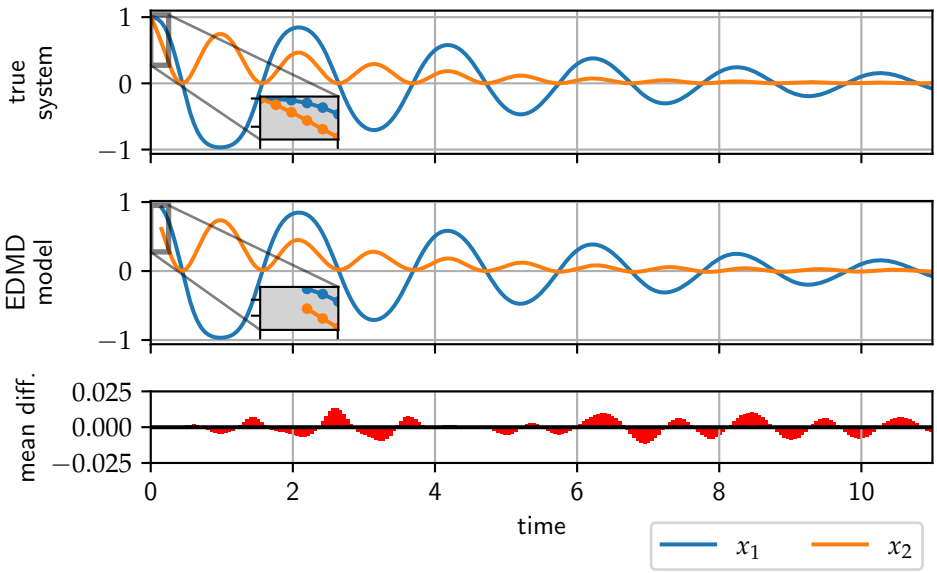


Figure 4.9: Comparison of the true (top) and predicted (middle) out-of-sample time series (Eq. 4.11) in Cartesian coordinates. The prediction is performed based only on the initial condition X_1. The included subplots highlight that the prediction starts later because the first three samples are required for the time delay embedding. The bottom row plots the mean value of the difference time series $\hat{\mathbf{x}}_j - \mathbf{x}_j$.

The model is able to predict the out-of-sample time series accurately and from the initial condition until it reaches the steady state (see the bottom row in Fig. 4.9). The predicted time series is computed in a single prediction loop — after the initial condition, the states evolve according to the linear system in Eq. 4.11. Given the pendulum's rod length of $l = 1$ the maximum difference in the two coordinates is relatively small with $\max\{\Delta x_1\} \approx 0.01$ and $\max\{\Delta x_2\} \approx 0.02$. While the phase and frequencies of the pendulum oscillations match between the true and predicted time series, the error still has an auto-correlated structure. This could suggest a need to further revise the model within the system identification loop. However, as highlighted in Section 2.3, it is important to acknowledge that even for simple systems an exact representation of the system in terms of the Koopman operator requires infinitely many observables, which is computationally intractable.

The Koopman triplet in Eq. 4.11 describes a discrete linear dynamical system with a sampling rate Δt of the underlying continuous system. In a matrix form of the system,

the Koopman modes are captured in $V \in \mathbb{C}^{[2\times 50]}$, where each column corresponds to a mode; Fig. 4.10 displays the first and last elements of each row. The Koopman modes remain constant in the model and reconstruct the measurement states from the Koopman eigenfunctions (cf. Fig. 3.1 on page 64). Another useful property is that the modes provide a way to sort the triplets based on their importance using the norm of each column in V; for details see Manojlović et al. [2020]. In the `EDMD` class, this feature is enabled with the flag `sort_koopman_triplets=True`.

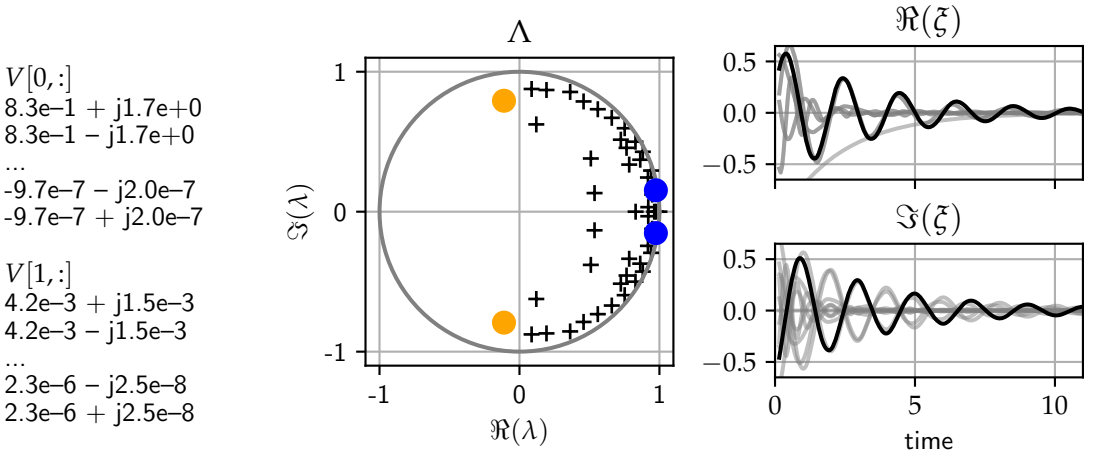


Figure 4.10: The Koopman triplet components are stored within the `EDMD` model for the identified pendulum system (based on the Cartesian coordinates). The modes V are left with the first and last elements of the matrix. The eigenvalues Λ are in the middle plot and scatter on the complex plane and a circle with unit length $|\lambda| = 1$. In the order of the triplets, the orange points mark the least important values and the blue points the most important. On the right side are the eigenfunctions plotted over time, where the real and complex parts of the function are plotted separately. The leading eigenfunction is in black and the following nine eigenfunctions are in gray.

The middle graph in Fig. 4.10 shows a scatter plot of the Koopman eigenvalues in the complex plane. The Koopman eigenvalues are the only time dependent quantities. This makes it easy to analyze the system behavior for each triplet term in Eq. 4.9. Because the pendulum is simulated with friction, the system is transient until it reaches the stable equilibrium point. This behavior is reflected by the eigenvalues, which are all inside the complex unit circle of radius one, $\left|\lambda_p^{\Delta t}\right| < 1$, and therefore decay towards zero over time. Important components are usually closer to the unit circle because they describe long term system behavior; In Fig. 4.10 the important eigenvalues in blue are $\lambda_{1,2} \approx 0.97 \pm j0.15$ with a magnitude of about 0.98. Ultimately, the selection of leading Koopman triplets also provides a way to build reduced-order models.

The last component of the triplet is the Koopman eigenfunctions. The two graphs show that the eigenfunctions exhibit harmonic functions and can contain geometrical information of the underlying system [Mezić, 2020]. The eigenfunctions are evaluated

along with the out-of-sample time series `X_test`; see the last statement in the above code snippet.

In summary, the approximation of the pendulum system was greatly improved by utilizing the geometrically adapted state representation within the EDMD. In contrast to the DMD model in Fig. 4.7, which only obtained a *local* approximation of easier-to-predict angular time series, the EDMD can perform a *global* system identification based on the harder-to-predict Cartesian time series. The EDMD dictionary contained the composed transformation (approximating the Laplace-Beltrami operator on reconstructed states), which showed high analytical power in the geometric analysis. In the dictionary, the function basis can be interpreted as a generalized Fourier basis [Belkin et al., 2009; Berry et al., 2013]. Note that the established connection of the two DMAP coordinates (ψ_2, ψ_6) to the hidden angular reference system in Fig. 4.5 – 4.6 makes it possible to predict these coordinates with a minor model adaptation. This means that the physically meaningful coordinates are recovered through manifold learning and the dynamics with the mode decomposition in the geometrically aligned coordinates.

The fact that only two time series in Cartesian time coordinates are required to generalize the model well for out-of-sample initial conditions is remarkable. Moreover, the `EDMD` is a linear model that comprises of "building blocks" that are founded in linear operator theory. These can give insight into the spatio-temporal patterns of nonlinear systems. I explore this aspect more in Section 4.3. The whole system identification workflow was performed with ***datafold*** using well-defined interaction points that align to the common machine learning interface from ***scikit-learn*** [Buitinck et al., 2013].

In the next two sections, I apply the approach to systems for which an analytic form of the system is not available.

4.2 Bus station: Surrogate model of a microscopic pedestrian simulator

This section draws on an example of time series data from the field of pedestrian dynamics: a simulation of pedestrian movement at a bus station. In the analysis, I build a coarse-grained surrogate model from time series data generated from a microscopic pedestrian dynamics simulator. The microscopic scale refers to a level where each pedestrian is modeled and simulated individually, whereas the macroscopic scale refers to aggregated and emerging quantities, such as the crowd density. The goal is that the data-driven surrogate model describes the specific system behavior independently of the underlying simulator. Fig. 4.11 gives a schematic overview of the procedure.

Within the data-driven modeling workflow, I transfer the main operator-informed approach and software implementation in ***datafold***. For the data application governing equations are no longer available. However, because the data can be simulated, it is possible to "design" the state quantities and systematically query the underlying simulator.

Pedestrian dynamics is an interdisciplinary research field that connects many disciplines, such as computer science, mathematics, sociology and psychology [Kleinmeier et al., 2019]. Simulation software for analyzing pedestrian dynamics draw on different aspects from these diverse fields. If the models are carefully calibrated and validated, the pedestrian simulator can become a scientific tool to perform "virtual experiments" and analyze a wide range of what-if cases in safety-relevant situations for which experimentation is impossible or too dangerous [Bungartz et al., 2014; Kleinmeier et al., 2019].

In the next Section 4.2.1, I detail the application setting of creating a surrogate model from a pedestrian simulator. In Section 4.2.2 I introduce the bus station scenario that I use as an example to generate time series data, which I then use as a basis for the data-driven surrogate model in Section 4.2.3. Finally, in Section 4.2.4, I highlight the applicability and computational benefit of the surrogate model in an uncertainty quantification (UQ) scenario where many model evaluations are required. UQ is an important tool of pedestrian dynamics because many parameter specifications remain uncertain in a concrete scenario even after calibration with real observations.

The results of this section are based on a conference presentation at the 2019 Traffic Granular Flow Conference in Pamplona, Spain and are published in the associated proceedings [Lehmberg et al., 2020a]. In this thesis, I take a slightly different modeling configuration and include additional aspects of cross-validation. Ultimately, this leads to a more accurate model.

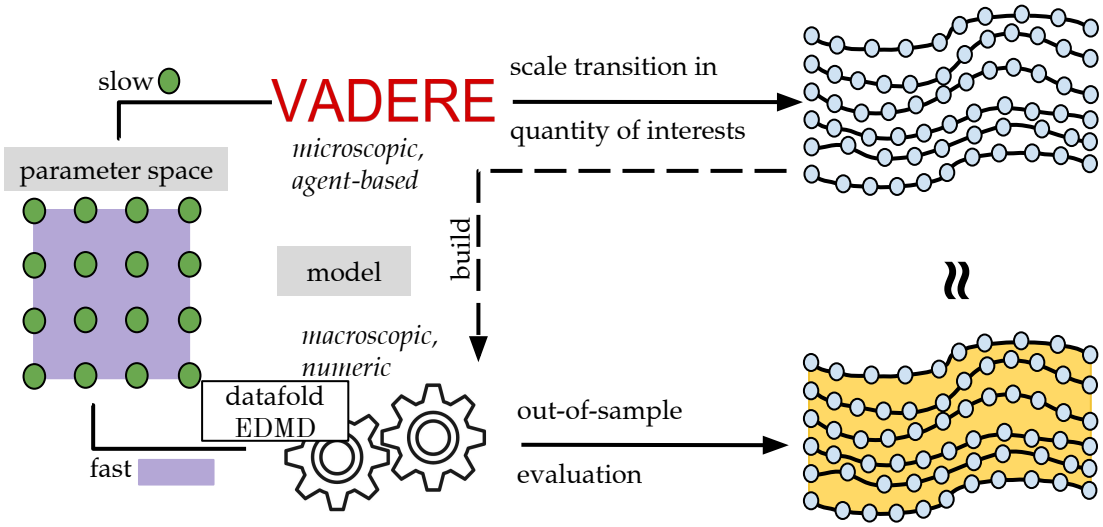


Figure 4.11: Schematic overview of a microscopic simulator (here *Vadere* – Kleinmeier et al. [2019]) and a surrogate model. The two models share the same parameter space. The generated example time series from the "slow simulator" is used as the basis for the surrogate model. The surrogate model can then efficiently interpolate the parameter space and generate time series. Adapted from Dietrich [2017, Fig. 3.1].

4.2.1 Data-driven dynamic surrogate model on a macroscopic scale

For the analysis of the bus station, I use the open-source software *Vadere*. The simulator describes the social behavior of pedestrians and topographic conditions in a *microscopic* and agent-based perspective [Kleinmeier et al., 2019]. This means that each agent — corresponding to a pedestrian — is simulated individually and the main modeling endeavor focuses on the microscopic update rules of an agent. In general terms *Vadere* classifies as a "complex system", in which many simulated components interact. Modeling at a microscopic scale is often favored in pedestrian dynamics because it is easier to describe individual human behavior in the explainable and accessible framework of agent-based update rules. This allows studying the emerging dynamical patterns of interacting agents. For example, the PhD thesis of Benedikt Kleinmeier [2021] describes how to add a psychology layer to locomotion models in *Vadere*.

Despite the pedestrians being modeled on a microscopic scale, the analysis and classification of pedestrian dynamics mostly focus on *macroscopic* quantities. This is because a microscopic state is very detailed, high-dimensional and hard to interpret, whereas crowd density or flow as examples of macroscopic quantities [Bode et al., 2019; Helbing et al., 2007] are much easier to comprehend. Moreover, characteristic macroscopic patterns can emerge from the collective interactions in real-world observations or simulations. Such macroscopic patterns are "more than the sum of the individual units" [Bonabeau, 2002]. This can be, for example, stop-and-go, laminar or turbulent crowd dynamics as well as transitions between these [Helbing et al., 2007]. Such patterns at multiple scales are common to many complex systems [Kevrekidis and Samaey, 2009]. The macroscopic patterns are therefore relevant to characterize pedestrian dynamics and validate simulators against these.

Constructing surrogate models via machine learning methods is an established approach to mitigate computational demands from microscopic simulators to perform sensitivity analysis or parameter studies [Niemann et al., 2021]. The idea is that the dynamics of the reduced macroscopic state are extracted from a specific scenario, by systematically sampling from a parameter space and collecting time series that describe quantities of interest. The parameter space is shared between the simulator and surrogate. But with the out-of-sample extension of the surrogate model, it is possible to interpolate the dynamics and directly generate time series that approximate the true dynamics of the original simulator. This is highlighted by the shaded backgrounds in the parameter space (purple) and the output time series (yellow) in Fig. 4.11.

Instead of evolving the computationally expensive microscopic state of a general purpose pedestrian simulator (here *Vadere*), surrogate models gain their computational advantage by directly evolving the reduced macroscopic state in a numerical model. An in-depth analysis of data-driven surrogate models to perform scale transitions is provided by Dietrich [2017] in his PhD thesis. Moreover, scale transitions in complex systems to reduce the computational burden of microscopic simulations are integral to the "equation-free framework" developed by Kevrekidis et al. [2003]. In a recent study by Niemann et al. [2021], the Koopman generator has been studied to construct reduced order surrogate models from agent-based systems with a variant of EDMD.

4.2.2 Data generation for bus station scenario

For the concrete traffic scenario, I model how pedestrians leave a bus and station exit. Fig. 4.12 displays a simulation snapshot and the topography as modeled in *Vadere*. When designing the scenario I favored the usefulness as a demonstrator for the surrogate modeling and follow-up analysis over realistic traffic parameters. Because the operator-informed approach that I apply is data-driven and does not require prior knowledge, it could easily be transferred to other scenarios or simulators (including commercial and closed-source ones).

Since real-world observations or experiments in pedestrian dynamics are often noisy due to unpredictable human factors and unknown forcing, *Vadere* includes various sources where random numbers are generated to anticipate observed variations in pedestrian dynamics patterns. This also affects the macroscopic quantities which become stochastic.

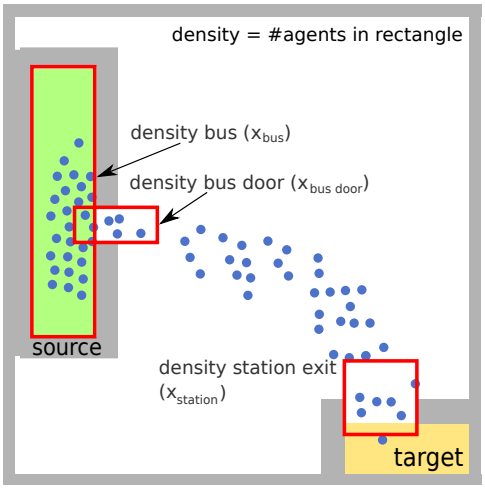


Figure 4.12: Snapshot of the modeled bus station scenario. The agents move from the source (bus, green) to the target (station exit, yellow) by avoiding collisions with the obstacles and other agents. The red rectangles highlight the regions in which the crowd density is measured. Adapted from Lehmberg et al. [2020a].

In Fig. 4.12, the simulated agents are displayed as blue circles, which can only navigate in the area that is not covered by an obstacle (in gray). At the initial simulation state, all agents are placed randomly with a uniform distribution inside the green rectangle, representing the interior of a bus. Furthermore, each agent has an individual free-flow speed, which is the speed at which an agent moves through an obstacle-free geometry. The free-flow speed of each agent is selected according to a truncated normal distribution, $\mathcal{N}_{\text{trunc}}(\mu = 1.34, \sigma = 0.26, \text{min} = 0.5, \text{max} = 2.2)$, which is the default setting of *Vadere*.

After the initialization phase, all agents navigate towards the yellow target region at the lower right corner, which represents the station exit. Once an agent reaches the target, it is removed from the simulation. *Vadere* provides a number of (competing)

locomotion models to update the agents depending on their individual properties (e.g. free-flow speed) and near surroundings [Kleinmeier et al., 2019]. For the bus scenario, I choose the Optimal Steps Model (OSM), developed by Seitz and Köster [2012] and updated in von Sivers and Köster [2015]. The OSM updates agents in an event-driven update scheme to mimic "natural" walking behavior. The next future position of an agent is obtained from an optimization of a utility function. The function is defined on the "walkable geometry" and includes terms of the navigation field to the target, obstacles or nearby agents; see Kleinmeier et al. [2019, Fig. 3]. For an in-depth analysis of simulated pedestrian navigation, I recommend the PhD thesis of Benedikt Zönnchen [2021].

When building coarse-grained surrogate models, a key problem is selecting a suitable set of quantities. While the application at hand often dictates quantities of interest, only choosing these can result in a loss of essential system information [Dietrich et al., 2016]. Selecting appropriate quantities often requires insight (or intuition) into the dynamics. The quantities in a system state can fulfill different tasks, such as (1) being relevant for the application (2) being useful to initialize a state and (3) capturing spatio-temporal patterns [Liu et al., 2014]. While the selection of type (1) is straightforward, the other two cases classify as "supporting quantities" to have a more informative state that leads to a more accurate surrogate model. If no guidelines (or intuition) for an appropriate state selection is available, Liu et al. [2014] describe an "equation- and variable-free" approach, which uses Diffusion Maps to automatically extract variables.

Within the bus scenario, I specify three regions in which I measure the crowd density by counting the number of agents for each time step (j). These three density values then correspond to a macroscopic state that is captured in time series states $[x_{\text{bus},j}, x_{\text{bus door},j}, x_{\text{station, j}}]$. Note that in Fig. 4.12 the two regions of the bus and door overlap. A single run with the scenario generates a time series with 300 time steps, where a single time step corresponds to $\Delta t = 0.2$ seconds in simulated time.

For the UQ analysis in Section 4.2.4, I use the number of agents in the bus at the start of the simulation as an uncertain parameter, which I denote as θ. Following Fig. 4.11 it is necessary to reflect this uncertain parameter also in the surrogate model. Here the state quantity x_{bus} makes the initialization easy to accomplish, because it gives a one-to-one correspondence between the uncertain parameter and the macroscopic initial state, $\theta = x_{\text{bus},1}$.

Since the surrogate model is data-driven and views the simulator as a black box, the next step is to systematically sample the parameter and collect the resulting macroscopic time series of each parameter sample. In Fig. 4.11 this is highlighted by the green dots that lead to the time series. For practical reasons, the simulator has to provide an interface to both systematically sample the parameter and retrieve results in an automated fashion. For *Vadere* this is provided by the Python package ***suqc*** (surrogate and uncertainty quantification controller, version 2.1), which I developed during my thesis and made openly available[1]. The ***suqc*** package provides an API to sample parameters of *Vadere* and collects the time series results in a ***pandas*** `DataFrame` [McKinney, 2011]

[1] www.gitlab.lrz.de/vadere/suq-controller

as an easy-to-process format. Moreover, it is possible to parallelize the sampling and average outputs over multiple simulations. While I am not detailing the software *suqc* here, I include code of the main function call to generate the data for the bus station scenario:

```
import suqc

# Specify parameter sampling of bus station scenario
# in Vadere to create example time series on macroscopic scale
vadere_sample = suqc.SingleKeyVariation(
    scenario_path="./bus.scenario", # vadere scenario file
    key="sources.[id==1].spawnNumber",
    values=np.linspace(10, 100, 20).astype(int),
    qoi="density.txt",  # read time series from from output file
    model=path_to_vadere_jar_file,
    scenario_runs=100, # run each sample multiple times
)

# run simulations and store density time series in vadere_result
overview_table, vadere_result = vadere_sample.run(njobs=-1)

# average the time series at each time step
vadere_mean = vadere_result.groupby(["id", "timeStep"]).mean()

# cast data to TSCDataFrame to process in datafold
X = TSCDataFrame(vadere_mean)
```

The uncertain parameter `spawnNumber` is sampled with 20 equidistant values between 10 and 100 agents at integer values. For the data generation, each parameter setting is sampled with 100 independent runs with a different random seed each. This is to average the stochastic effects in *Vadere* and obtain mean simulation results for the traffic scenario. For the data collection I therefore perform $20 \cdot 100 = 2000$ simulation runs in total. All runs are in parallel and each run produces a time series, which is stored in a `density.txt` file. Once all time series are available in `vadere_results`, the runs for each parameter are averaged to a single time series in `vadere_mean`. The final statement in the above code snippet is a simple cast to *datafold's* time series collection format (`TSCDataFrame`) and can now be used as a basis for the operator-informed surrogate model. The workflow allows the traffic scenario and other specifications in *Vadere* to be easily exchanged.

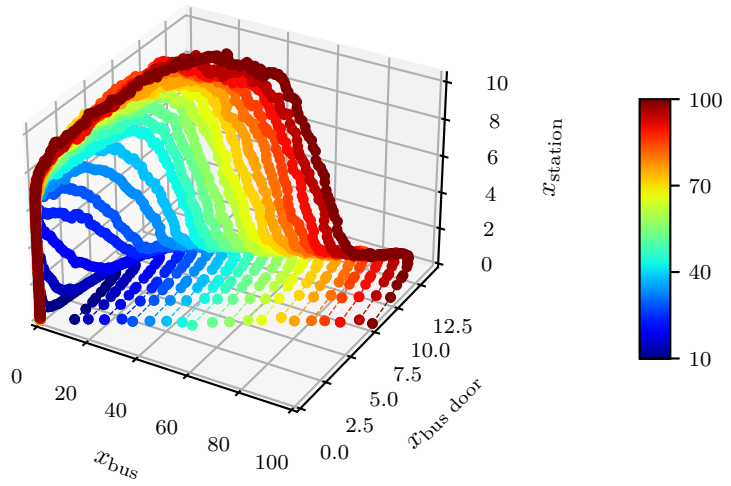


Figure 4.13: Averaged time series as generated and aggregated from *Vadere* as the microscopic simulator. The color code corresponds to the initial number of agents in the bus $x_{\text{bus},1}$. Adapted from Lehmberg et al. [2020a].

The final generated and averaged time series of the three density values are displayed in Fig. 4.13. Note that the system is in a transient regime most of the time, where states converge to the origin as the attractor state $[0, 0, 0]$ in which all agents have left the scenario.

The goal is to use the time series data and build a surrogate model, that both reconstruct the time series from *Vadere*, but also generates new time series based on initial conditions not included in the sampling. An important aspect for the modeling is the *closure* of the macroscopic dynamics [Dietrich et al., 2018]. That is, that the states are well-defined and clearly map to a future state. As highlighted in Section 2.4.2, a common way to reconstruct ill-defined state dynamics is time delay embedding. However, here I take a slightly different approach, by augmenting a supporting "slack quantity" to the state. I compute this slack quantity with

$$x_{\text{slack},j} = x_{\text{bus},1} - x_{\text{bus door},j} - x_{\text{station},j}. \tag{4.12}$$

The additional quantity is similar to a time delay embedding: In a geometric picture, it stretches states apart which are nearby but originate from time series with different initial states. Instead of ending at the origin as a single state, with the additional slack quantity the time series now end on an attractor line $[0, 0, 0, -x_{\text{bus},1}]$. Note that $x_{\text{bus},1}$ corresponds to the uncertain parameter and is a constant term in Eq. 4.12.

The final time series collection is then

$$X^{(i)} = [\mathbf{x}_1^{(i)}, \mathbf{x}_2^{(i)}, \ldots, \mathbf{x}_{300}^{(i)}] \in \mathbb{R}^{[4\times 300]} \tag{4.13}$$

$$X = [X^{(1)}, X^{(2)}, \ldots, X^{(20)}], \tag{4.14}$$

where each single time series $X^{(i)}$ is represented as a matrix and horizontally stacked to the final matrix X. Each snapshot contains the final state quantities in a column vector, $\mathbf{x}_j^{(i)} = [x_{\text{bus}}, x_{\text{bus station}}, x_{\text{station exit}}, x_{\text{slack}}]_j^{(i)}$ (i-th time series with time index j). Eq. 4.13 is therefore the data basis for the surrogate model, which directly captures the bus station dynamics on the macroscale.

4.2.3 Building data-driven dynamic surrogate model

With the generated time series collection of Eq. 4.13 it is now possible to construct a data-driven surrogate model, approximating the macroscopic system dynamics of the bus scenario. This means the surrogate model describes the flow $F_{\Delta t}$ of the three density values in Fig. 4.13 and the slack variable contained in a state $\mathbf{x}$,

$$\mathbf{x}_{j+1} \approx F_{\Delta t}(\mathbf{x}_j; \theta). \tag{4.15}$$

Because the time series data and process include noise, the flow describes the *expected* state evolution. Importantly, the model in Eq. 4.15 should not only reconstruct the generated data itself, but also generalize to new initial conditions $\mathbf{x}_1 \notin X$. For the bus scenario, the initial state $\mathbf{x}_1$ is specified according to the initial number of agents in the bus (in *Vadere*: parameter `spawnNumber`). For the surrogate model the following map is set up:

$$\mathbf{x}_1(\theta) = [\theta, 0.08 \cdot \theta, 0, 0, x_{\text{slack},1}]^T, \tag{4.16}$$

where $\theta \in (10, 100)$, corresponding to the sampling bounds. Given the measurement regions in the bus scenario of Fig. 4.12, the second quantity ($x_{\text{bus door}}$) includes a factor 0.08, which corresponds to the mean number of people inside the bus door region at initialization. The slack variable is computed according to Eq. 4.12 To identify the system dynamics of the macroscopic density values, I use the EDMD framework to approximate the Koopman operator and the implementations in *datafold* (Sections 2.3 and 3.5). For the EDMD dictionary, I approximate the first P eigenfunctions of the Laplace-Beltrami operator $\{\psi_p(\mathbf{x})\}_{p=1}^{P}$ with Diffusion Maps (Sections 2.4.2 and 3.4.3). In *datafold* the data-driven operator-informed surrogate model is set up in the following way:

```
edmd_base = EDMD(dict_steps=[
                ("dmap", DiffusionMaps(
                            GaussianKernel(epsilon=1),
                            n_eigenpairs=10))
            ],
            include_id_state=False)
```

A major property of this model setting is that it is equation-free and does not include any *a priori* knowledge of the bus scenario itself. Instead the Diffusion Maps

(DMAP) coordinates are extracted from the data geometry. This makes the model configuration easy to transfer to other settings with different measurement states (e.g. flow or pedestrian streams). Note that in my own publication Lehmberg et al. [2020a], I used the related geometric harmonics functions [Coifman and Lafon, 2006a] instead of DMAP coordinates. At the time the essential out-of-sample extension was missing in the `DiffusionMaps` implementation. In comparison to the publication, I can further improve the model quality in the thesis by using the DMAP coordinates and systematic parameter optimization.

Mathematically, the final system can be described in terms of the Koopman triplet $(V, \Lambda, \xi(\mathbf{x}))$ and the EDMD dictionary $\mathbf{g}_{\text{dmap}}(\mathbf{x}) = [\psi_1(\mathbf{x}), \ldots, \psi_P(\mathbf{x})]$ (see Eq. 2.44 on page 45):

$$\hat{\mathbf{x}}_{j+1} = V\Lambda^j \xi(\mathbf{x}_1(\theta)) = V\Lambda^j \Phi^{-1} \mathbf{g}_{\text{dmap}}(\mathbf{x}_1(\theta)). \tag{4.17}$$

The $\hat{\mathbf{x}}$ are predicted states and the matrix Φ^{-1} includes the left eigenvectors (by convention row-wise oriented) of the Koopman matrix diagonalization, $U_{\Delta t} = \Phi\Lambda\Phi^{-1}$ (see Eq. 2.26 – 2.29).

Because the model is non-parametric — increasing the number of DMAP functions P also increases the model complexity — it is important to optimize the model's parameters to avoid overfitting. For the above `EDMD` specification, I choose to optimize the bandwidth of the Gaussian kernel (`epsilon`) and P as the number of leading DMAP coordinates to include in the EDMD dictionary (`n_eigenpairs`). For this I make use of `EDMDCV` in *datafold*:

```
# median value of pairwise distances in X
m = 1360.5066

# set up parameter optimization in a grid search
edmdcv = EDMDCV(
    estimator=edmd_base,
    param_grid={
     "dmap__kernel":
       [GaussianKernel(epsilon=eps)
        for eps in [m/2, m, m*2, m*3, m*4, m*5, m*6]],
     "dmap__n_eigenpairs": [100, 120, 140, 160, 180, 200, 220],
    },
    cv=TSCKfoldSeries(5, shuffle=True, random_state=1),
    n_jobs=-1,
    refit=True
)

# run the optimization with the training data
edmdcv.fit(X)
```

```
# obtain final EDMD model
edmd_final = edmdcv.best_estimator_
```

`EDMDCV` performs a naive search on a parameter grid, as covered in Section 3.5.2 and Fig. 3.15. The best parameter specification for the final model is the pair with the lowest mean error (or highest mean score) in the validation data. For both parameters I specified seven values, which describe a grid with $7 \cdot 7 = 49$ candidate pairs. Note that the optimal kernel bandwidth (`epsilon`) depends on the sample density in the dataset. I therefore describe the parameter in terms of the median of the pairwise squared Euclidean distances in the training dataset

$$\varepsilon = c \cdot \text{median}(\|\mathbf{x}_i - \mathbf{x}_j\|)_2^2, \forall (i,j) \in X_{\text{train}}. \tag{4.18}$$

The relative setting of ε provides a good starting value for different datasets with different pairwise distances and hence makes it easier to transfer the model.

For each parameter sample, the time series collection is allocated in different training and validation configurations. Because the system mostly shows transient dynamics, it is best to perform splits along the separate time series (cf. Fig. 3.6 on page 82). In contrast, splits along the time axis would always exclude essential parts of the state space and require the model to extrapolate these missing validation parts. As per Fig. 4.11, for the surrogate model, I am only interested in generating time series that interpolate the state space. The specified `TSCKfoldSeries` class in the above code snippet performs five different configurations, such that 16 time series are used to train the model and four are used for the validation. In total $5 \cdot 49 = 245$ model fits are performed for the optimization plus one final model for the entire dataset.

Fig. 4.14 displays the reported errors of `EDMDCV` for both the training and validation error. The marked squares denote the parameter combination with the respective lowest mean error. The red color in the left training square shows that the errors are almost indistinguishable. The best parameter setting is located in the lower left corner, suggesting that further decreasing the kernel bandwidth ε and increasing the number of eigenpairs P could decrease the error. This shows that the two parameters steer the model's complexity because the training error can be "arbitrarily" decreased. However, there is a danger to choosing a parameter setting that overfits the training data.

The validation error in the right plot of Fig. 4.14 is suitable to balance the complexity and find a parameter pair that also adequately reconstructs time series that are not available to the model. In contrast to the reported training error, the plot shows that there is more variation in the validation error and that the *largest* error rates are located in the lower left corner. This highlights that the model indeed overfits at the lowest training error.

To the best of my knowledge, such model selection and validation analysis is currently missing in the literature for Koopman operator-based methods. There is also no software available that would support such an analysis for the EDMD framework. Because it is easy to fall into the trap of only basing the reconstruction optimization on available data, it is likely that often overly complex models are reported which also

include non-essential patterns such as measurement noise and would perform poorly for out-of-sample data.

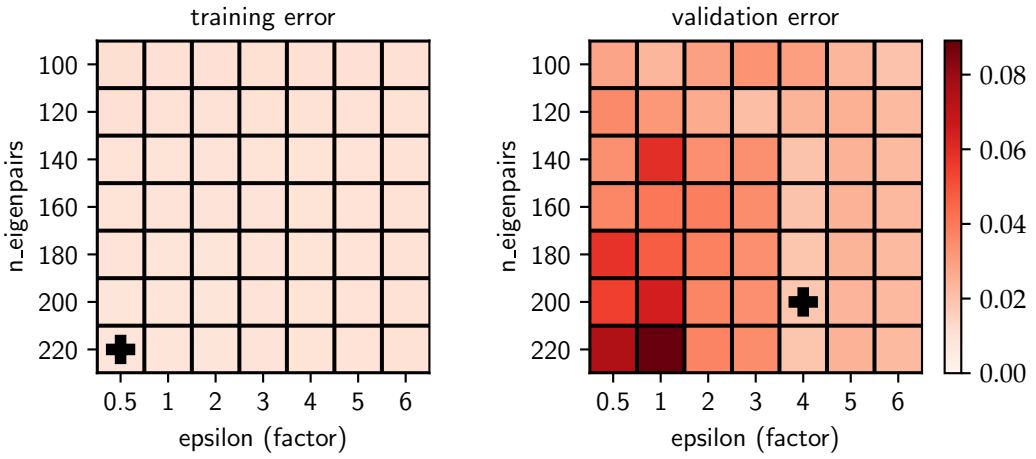


Figure 4.14: Visualization of the error rates in the parameter grid in `EDMDCV` for the training (left) and validation set (right). A black cross marks the best candidate pair in a grid. Note that the `epsilon` is in terms of the factor c of Eq. 4.18.

For the analysis, I proceed with the model that is fitted to the entire data set and parametrized with the lowest validation error. According to Fig. 4.14 the final parameter selection is `epsilon=5442.0264` and `n_eigenpairs=200`. With this parameter set, I accept a worse reconstruction error to obtain a more accurate generative model.

Given the interface of the `EDMD` model, the data of Eq. 4.13 can easily be reconstructed with the surrogate model:

```
X: TSCDataFrame
X_reconstruct = edmd_final.reconstruct(X)
```

Fig. 4.15 compares the original data from *Vadere* and the reconstructed time series with the surrogate model. The figure separates each density quantity into a dedicated column. It is apparent that the three density quantities have different spatio-temporal patterns and state evolution. The more agents that are in the bus at the start of the simulation, the longer the duration of (high) crowd density at the bus door and station exit.

While for all initial conditions the density increases at around the same time (at $j = 1$ for the bus and about $j = 50$ for the station exit), the location of peaks with high density and the slope for increasing or decreasing density vary. For example, at the bus door, all agents move immediately to the measurement region, such that the highest density peaks appear shortly after the simulation starts. In contrast, the peaks in density at the station exit appear later and are lower, which is a result of a wider exit door compared

to the narrow bus door. Note that the color-coded density is relative to the maximum observed value in the measurements.

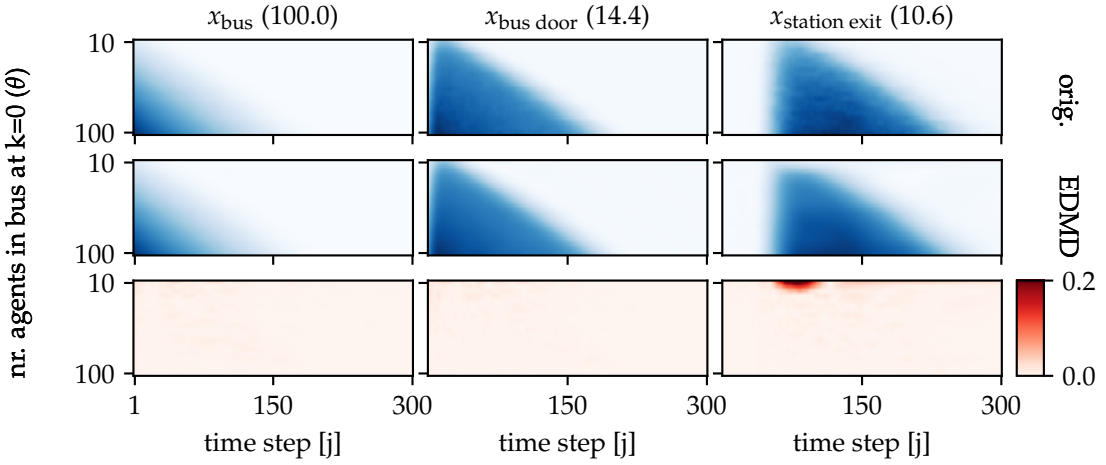


Figure 4.15: Comparison of the data generated by the simulator (top row) and the reconstructed time series with the surrogate model (middle row). The color code in the first two rows is relative to the maximum observed value highlighted in brackets. The bottom row highlights the difference values of the two plots.

All patterns in Fig. 4.15 are distinct and specific to the simulation settings, such as the topography and the locomotion model. While the original crowd density in X are only accessible through *Vadere*, the Koopman operator-based surrogate model makes it possible to describe and interpolate the patterns that emerge from the microscopic state. All quantities and dynamics are captured in the EDMD dictionary and a Koopman operator-based linear dynamical system; Eq. 4.17.

The third row in Fig. 4.15 shows the relative error between the surrogate model and simulated data. In general, the error is relatively low for all three quantities, which suggests a good fit of the surrogate model. The largest error occurs at the station exit for the initial conditions of 10 to 15 agents at times of positive density. A possible reason is that low densities have greater relative fluctuations in the time series. Smaller numbers of agents in the scenario make it harder to transition to a positive density in the station exit. The reconstruction error analysis is therefore valuable because it points to problematic state dynamics.

I want to highlight that the surrogate model is described in terms of oscillatory functions, both in the dictionary and Koopman eigenfunctions. This means that the model can also have (slightly) negative density values. These unrealistic values could easily be removed by setting them to zero. However, I choose not to further manipulate the surrogate model with knowledge of system constraints.

The last important factor is the out-of-sample extension to interpolate the state space. For the forward UQ analysis exemplified in Section 4.2.4, the generative property is essential to define probability distributions on the uncertain parameter; here θ as the

initial number of agents in the bus. The EDMD dictionary contains the out-of-sample mapping in DMAP, which is performed by the Nyström extension (cf. Eq. 2.42 and 2.43 on page 45).

Fig. 4.16 plots the three density values of the modeled system by interpolating the states both in space (by including more initial conditions $\theta \in [10, 100]$) and time (by decreasing the sampling rate $\Delta t < 0.2$). Instead of only having time series as in Fig. 4.13, which only form a point cloud and no explicit governing equations, the surrogate model now captures the full identified state space with an explicit linear system (Eq. 4.17).

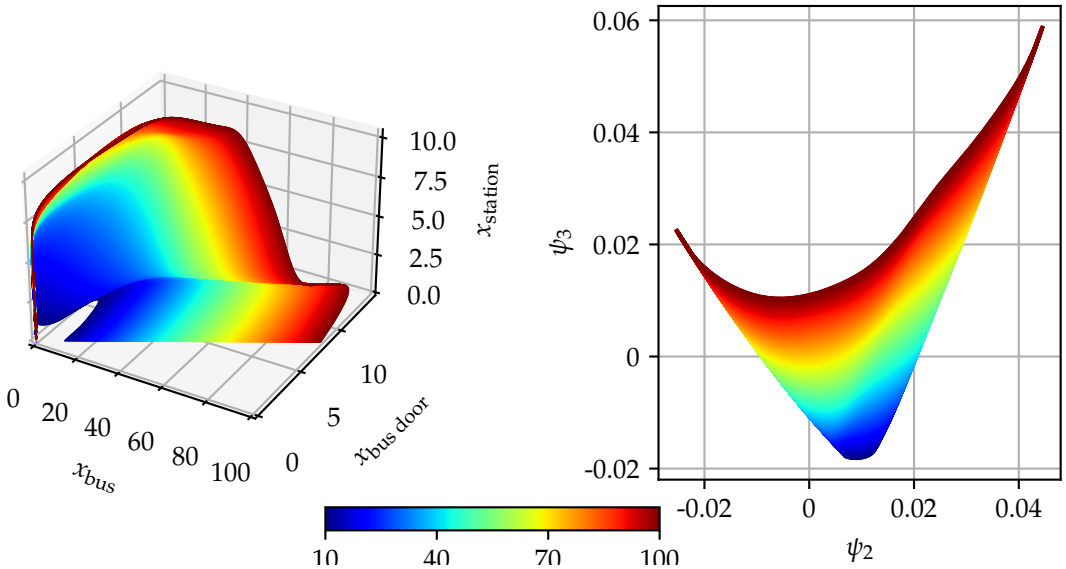


Figure 4.16: Left: The interpolated state space (only the density quantities) generated with the surrogate model by interpolating the initial condition $\mathbf{x}(\theta)$ with $\mathbf{x}(\theta) \in [0, 100]$. Right: The first two non-trivial DMAP coordinates, which describe geometrically independent directions of the inferred state space manifold. The color-code corresponds to the initial number of agents in the bus.

Because the DMAP coordinates have their origins in manifold learning (unsupervised machine learning) [Coifman and Lafon, 2006b], they also expose interesting patterns of the identified system and can help to better understand the macroscopic dynamics, which are only indirectly described through the microscopic simulator. The intrinsic spectral coordinates in a geometric perspective can therefore be used to compare different locomotion models or validate models with real-world observations. Ultimately, this leads to an interesting future direction to match dynamical systems, for which the spectrum in the Koopman operator is particularly well-suited, see Bollt et al. [2018].

As also demonstrated for the pendulum system in Section 4.1, the DMAP coordinates can describe a geometry that relates to the underlying state space. The right plot

of Fig. 4.16 displays the first two coordinates (after the trivial first constant coordinate), $[\psi_2(\mathbf{x}), \psi_3(\mathbf{x})]$. The shape resembles a curved triangle, where the lower end (blue) describes short time series with a small number of agents in the bus and the higher end (red) parametrizes time series with a maximum number of agents in the bus. All initial conditions (parametrized with $\theta \in [10, 100]$) lie on the right edge of the geometry and move towards the attractor line on the left edge. Note that the DMAP coordinates are computed from the entire state, which also includes the slack variable as a fourth quantity that makes the attractor in the measurement space a line. In the modeling approach of Dietrich [2017] these intrinsic coordinates are used to construct a nonlinear surrogate model since these coordinates parametrize a qualitative copy of the underlying state space manifold.

The `EDMD` model also stores the (approximated) Koopman triplet as system-intrinsic components that provide insight into the modeled dynamics. However, in the analysis of the surrogate model my main objective is to have a model that efficiently evaluates time series that accurately approximate the underlying system. For an extended analysis to gain insight into the identified system using real-world data, I refer to Section 4.3.

4.2.4 Forward uncertainty quantification

I now show how the constructed surrogate model can be used in a forward uncertainty quantification (UQ) analysis. Like with the bus scenario itself, the main purpose of this analysis is to highlight the applicability of the surrogate model and show how the methodology supports parameter studies. In particular, I showcase the computational advantage once the surrogate model is available.

Fig. 4.17 highlights the setup for the forward UQ analysis, where a probability distribution is defined on the uncertain parameter θ (the initial number of agents in the bus). This uncertainty is carried through the model and affects the uncertainty in the observed density time series. UQ is a relevant topic in pedestrian dynamics because the microscopic models have many parameters that cannot be measured or have no one-to-one correspondence to real observations. For example, the OSM includes repulsive and attractive forces to avoid collisions between the agents and obstacles [Seitz and Köster, 2012]. The methodology of UQ allows a simulator to be analyzed under uncertainty, where the outcome of a simulation is no longer a point evaluation but a distribution in which the likelihood of worst-case or best-case outcomes can be deduced. von Sivers et al. [2016] use UQ to analyze the effect of a parameter that describes the social identity. In a similar setting to the bus station, Dietrich et al. [2018] build a dynamic surrogate model to perform UQ in an egress scenario at a train station. Florian Künzner [2020] investigates non-intrusive UQ for large scale simulations (including pedestrian dynamics) in his PhD thesis.

The uncertain number of agents in the bus means that the entire evolution of the resulting time series is also uncertain. Fig. 4.17 highlights that there is an output dis-

tribution at each time value for all three density values over time. In the analysis, I compare the original simulator *Vadere* with the built EDMD surrogate model.

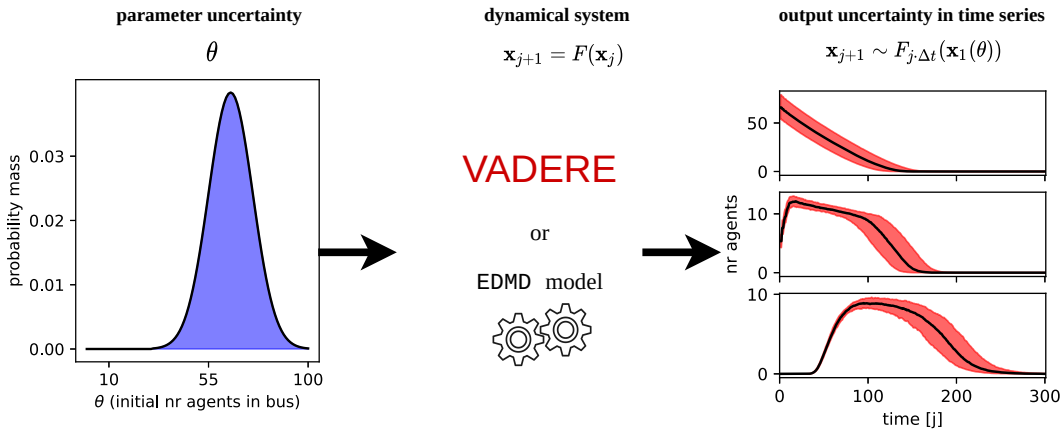


Figure 4.17: Schematic representation of the forward uncertainty quantification setting. Starting on the left, the parameter uncertainty is described with a probability distribution. The objective is to observe the effect of the uncertainty θ on the output of the multivariate time series on the right. The uncertain output is described with the median (black line) and the (10%, 90%) quantiles (red band). To obtain the relation between input and output uncertainty either the microscopic simulator or the surrogate model (acting as dynamical systems) can be sampled.

For the forward UQ analysis I specify three different (representative) probability distributions on the uncertain parameter, which now acts as a random variable:

$$\theta_1 \sim \text{Uniform}(\text{loc} = 40, \text{scale} = 20) \tag{4.19}$$
$$\theta_2 \sim \text{TruncatedNormal}(\text{a} = -3.5, \text{b} = 3.5, \text{loc} = 65, \text{scale} = 10) \tag{4.20}$$
$$\theta_3 \sim \text{TruncatedExponential}(\text{b} = 100, \text{loc} = 20, \text{scale} = 10). \tag{4.21}$$

To describe the distributions I use the arguments of the respective functions in the statistical module of the *SciPy* package [Virtanen et al., 2020].

The forward UQ is performed by drawing samples from the distributions in a Monte Carlo fashion. To compare the microscopic simulator *Vadere* and the surrogate model, I generate a time series for each draw of θ respectively. A practical aspect is that microscopic simulators can only be parametrized with integer values of θ, while the surrogate model has the capacity to also interpolate real-valued states (cf. Fig. 4.11). To facilitate direct comparison between simulator and surrogate, I truncate all θ values to the next integer, making the distributions essentially discrete.

Fig. 4.18 shows the uncertain output of the three distributions. The surrogate model approximates the true output distribution over time well, as all median curves match satisfactorily. The largest discrepancies are at the lower 10% quantile of the station exit.

This is a result of the higher error rates observed in the surrogate model for regimes with a small number of agents (cf. Fig. 4.15). Much of the probability density of the truncated exponential (θ_3) lies in this regime. This highlights the importance of the reconstruction error analysis. Furthermore, the truncated exponential distribution also shows the largest variance and reflects the non-symmetric input in a skewed output uncertainty.

In Fig. 4.18 the evolution of the uncertainty over time shows narrow bands during the initial density increase; the uncertainty instead manifests in the output during stagnant and decreasing density regimes. The surrogate model is able to capture these characteristics and separate the different cases over time. Note that the whole simulation is only based on the single state $\mathbf{x}_1$.

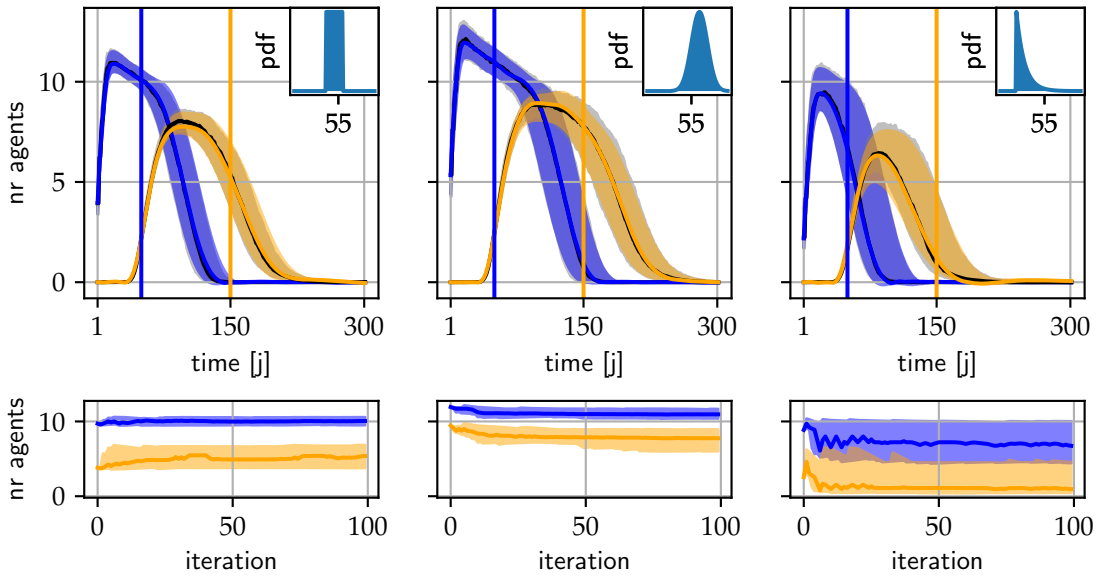


Figure 4.18: Top row: The plots show the output uncertainty over time, each resulting from the respective input parameter probability distribution functions (pdf) from Eq. 4.19 – 4.21 (also shown in the inset in each plot). Only the density values at the bus door (blue) and the station exit (orange) are displayed. The curve corresponds to the median and the band describes the interval of the (10%, 90%) quantiles. The gray band and black curve in the background is the output uncertainty from *Vadere*. Bottom row: The convergence rates of the quantiles, which are incrementally computed for each new Monte Carlo iteration. The Monte Carlo convergence is measured at a single time step, marked with vertical lines at $j = 50$ for the bus door and $j = 150$ for the station exit.

Because of the linear regression and truncation of DMAP coordinates in the EDMD model, the surrogate model shows smoother bands compared to the noise corrupted uncertainty bands (gray) of the original simulator. Similar plots for a forward UQ analysis of pedestrian dynamics are also presented in von Sivers et al. [2016, Fig. 13] and Dietrich et al. [2018, Fig. 7].

For the computational comparison below, I made sure that I drew sufficiently many Monte Carlo draws from the distributions such that the quantiles converge. The second row in Fig. 4.18 includes the convergence rates of the quantiles at a specific time value. The figures show that all quantiles converge after 100 samples. In more complex scenarios with a larger set of uncertain parameters, the number of samples can be much higher.

Table 4.1 compares the computational time between the simulator and the surrogate model. All Monte Carlo evaluations of the bus scenario were executed in parallel. However, for an easier comparison, the table includes the summed wall clock time of all single evaluations. Note that obtaining a single time series from *Vadere* also requires 100 runs to average the random fluctuations. For the 100 random samples in *Vadere* this makes a total of $100 \cdot 100 = 10000$ runs for each distribution.

Table 4.1: Runtime between *Vadere* (microscopic pedestrian simulator) and the EDMD surrogate model. The tasks distinguish between "offline tasks" for model construction and "online tasks" in which the UQ analysis is performed. Because *Vadere* is the base to construct a surrogate model, the offline tasks are not applicable (n/a). All numeric values are in seconds.

Task		***Vadere***	**EDMD**	
1	generate time series with *Vadere*	n/a	37913	offline
2	model selection with `EDMDCV`	n/a	27355	
3	forward uniform distribution Eq. 4.19	190,039	0.296	online
4	forward truncated normal distribution Eq. 4.20	214,268	0.285	
5	forward truncated exponential distribution Eq. 4.21	167,421	0.278	
	sum	571,728	65,268	

As per Table 4.1 the construction and evaluation of a surrogate model separates between an offline and online phase (see also Dietrich et al. [2018]). The first row describes the generation of the crowd density example data with *Vadere* (Section 4.2.2) and the second row includes the model fit and error evaluations in the parameter optimization with `EDMDCV` (Section 4.2.3). The rows three to five then include the runtimes to perform the forward UQ evaluation for each of the three distributions.

In the online phase, the computational advantage of the surrogate model compared to *Vadere* is immense. Because the surrogate bypasses the microscopic evaluations and averaging routines, it only requires a tiny fraction of what is needed to perform an exact evaluation with microscopic simulations.

Another positive aspect of the surrogate model is that the state evaluations require a constant number of floating point operations in Eq. 4.17. This means that the model evaluations remain approximately constant irrespective of the uncertain parameter distribution (θ_{1-3}). In contrast, the memory requirements and runtime of *Vadere* are influenced by the microscopic state and, therefore, how many agents are simulated. For

example, the truncated exponential distribution — having a higher probability density at smaller numbers of agents — takes only around 78% of the runtime compared to the truncated normal distribution.

A fast evaluation and more predictable runtimes make the surrogate model approach a great candidate in applications where rapid (or even real-time) evaluations and decisions are required, see also Dietrich et al. [2018] and Niemann et al. [2021]. The surrogate model can also be used when the parameter's uncertainty distribution changes, as highlighted in Table 4.1. For example, this would be the case in the bus scenario if the parameter uncertainty changes during the day depending on the current traffic. This contrasts with other common approaches for surrogate models in UQ, such as the Polynomial Chaos Expansion (PCE). While PCE also limits the number of evaluations of the simulator under study, as soon as the parameter distribution changes new example data must be generated from the simulator to construct a new surrogate model [Dietrich et al., 2018].

To conclude the analysis in this section, I could demonstrate the suitability of the Koopman operator approach to construct an efficient and accurate surrogate model. The model performed a scale transition of a microscopic simulator to the macroscopic crowd density patterns.

The model is data-driven and requires no specific *a priori* knowledge and can therefore easily be transferred to other (traffic) scenarios. Furthermore, I could capture the nonlinear spatio-temporal patterns of multiple quantities of interest in a single Koopman matrix describing a linear system. The intrinsic geometric coordinates are suitable for analyzing the identified systems. I will take advantage of these factors in the next section, where I analyze real-world sensor measurements.

An interesting direction for future work would be to limit the number of microscopic evaluations by adopting "active learning" and to vary the sampling rate of the parameter space according to the "complexity" of the resulting time series.

4.3 Melbourne sensors: Gaining insight into pedestrian dynamics

For the data scenarios in this section, I again perform system identification on collections of time series that originate from pedestrian traffic. However, here I extend the established operator-informed setting and *datafold* to *real-world* sensor measurements, provided by the City of Melbourne [Melbourne, 2021a]. There are fundamental differences in the underlying system and data characteristics between the previous two scenarios and the sensor-based pedestrian data analyzed here: (1) the states are not converging to an attractor point, (2) the governing equations of the system are unknown, and (3) the states are high-dimensional and disturbed by many unobserved factors. Most of the results covered in this section are also published in a journal article [Lehmberg et al., 2021].

In a broader context, the analysis contributes to mobility research of "smart city" systems. In such settings a network of sensors across a city measure and stream traffic-related quantities to central units [Carter et al., 2020]. Collectively, all measurements describe a traffic state on a city scale. Global monitoring can assist authorities to improve traffic and public safety [Carter et al., 2020; Nagy and Simon, 2018]. A key objective is to describe and forecast the (pedestrian) traffic, because this helps to allocate limited security resources effectively and identify traffic abnormalities [Zameni et al., 2019].

In contrast to the simulated traffic scenario in the previous section, the direct data-driven modeling on real-world observations is advantageous for predicting future traffic at sensors. While simulators such as *Vadere* for pedestrians [Kleinmeier et al., 2019] or *SUMO* for vehicular traffic [Lopez et al., 2018] are useful to understand complex interactions and perform virtual experiments in what-if scenarios, a forecast based on macroscopic sensor observations is usually not possible. This is because for large scenarios on a city scale, setting up a simulation requires laborious customization [Nagy and Simon, 2018]. A more serious problem is that the initialization of a microscopic state in a simulator is ill-defined. The aggregated and macroscopic system information is insufficient to specify a distinct microscopic state in the simulator. For example, in the Melbourne sensor data used here, the origin, destination and walking speed of an individual pedestrian — mapping to an agent as part of the simulation — are unknown. The ill-defined map from an (aggregated) macroscopic state to a (detailed) microscopic state is a common problem that is also highlighted in the equation-free framework by Kevrekidis and Samaey [2009].

This introduction continues with a short review of previous research using machine learning approaches, including the Koopman operator methodology, to analyze traffic systems. I then describe the characteristics and selection of the Melbourne sensor data in Section 4.3.2. In Section 4.3.3 I outline the main model architecture and parametrization to identify the traffic system. In Section 4.3.4 I show that the modeling approach under these circumstances leads to an accurate model to collectively forecast all included sensors. In the subsequent analysis of Section 4.3.5, I use the model's components to gain insight into model characteristics and analyze the intrinsic geometric and dynamic coordinates of the identified system. Since these are based on the operator theory, this provides a path to enhance both the predictive capabilities of complex systems and the scientific understanding of traffic systems.

4.3.1 A short review of related research

Research has utilized a large array of machine learning models to analyze real traffic systems; for review articles I refer to Boukerche and Wang [2020]; Nagy and Simon [2018]; Vlahogianni et al. [2014]. These methodologies include common data-driven approaches, which are highlighted in Section 2.2.3. Popular techniques are Support Vector Regression (SVR) [e.g. Yin et al., 2020] or Autoregressive Integrated Moving Average (ARIMA) as statistics-based methods [e.g. Boukerche and Wang, 2020; Nagy and Si-

mon, 2018] and deep neural networks (DNNs) architectures [e.g. Li et al., 2020; Lim and Zohren, 2021; Lv et al., 2014; Vlahogianni et al., 2014; Yao et al., 2019]. A system identification model extracts data-intrinsic correlations based on historic traffic states. This approach therefore complements the ever increasing data availability stemming from widespread and inexpensive sensors installed in cities. The interaction between the modeled patterns then determine the forecast of a traffic state [Boukerche and Wang, 2020].

While the aforementioned methods are applied more frequently, the Koopman operator theory has also been introduced recently to traffic analysis. Avila and Mezić [2020] analyze real-world data from a multi-lane highway. Similar to my analysis they show how the operator-based approach uncovers intrinsic spatio-temporal traffic patterns. Their model can be used for both system analysis and traffic forecasting. Liu et al. [2016] explore the DMD for vehicular highway traffic, where they detect temporal patterns on different time scales.

For pedestrian traffic, Benosman et al. [2017] estimates crowd flow based on incomplete spatial measurements. In Dicle et al. [2016] the DMD-variants are useful to analyze a system's stability by only processing noisy and high-dimensional data from pedestrian video streams. Finally, Cheng et al. [2021] show that it is possible to use a DMD-variant from [Le Clainche et al., 2017] to forecast a origin-destination matrix, describing pedestrian transitions in a metro station.

4.3.2 Data description and selection

For the real-world dataset, I use pedestrian traffic data from sensor measurements provided by the City of Melbourne, Australia. The full dataset and sensor descriptions are freely available [Melbourne, 2021a,b] [2]. Fig. 4.19 shows an example of a single sensor setting. The sensors are installed on awnings or street poles at main pedestrian thoroughfares or areas with high retail or event activities. A single sensor records the number of pedestrians that passed under that sensor in the past hour with a timestamp on the hour. Together all sensors describe a traffic state, which comprises all measurements at a given time. I detail the final data selection used for the data analysis in Section 4.3.2.

[2] An interactive city map with visualized sensor measurements and locations is provided at http://www.pedestrian.melbourne.vic.gov.au/.

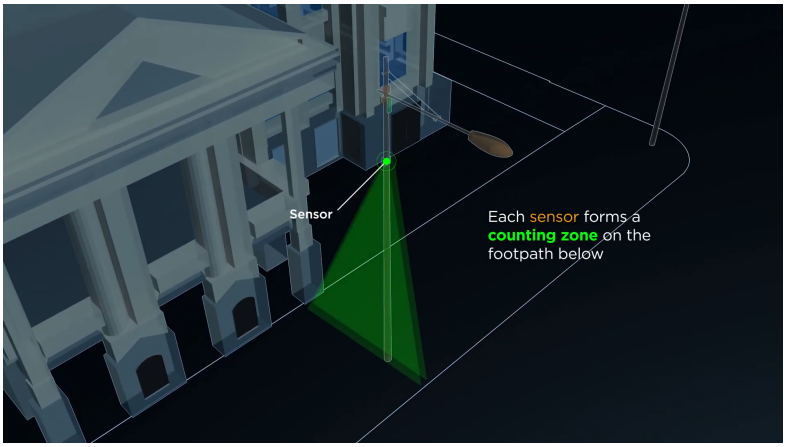


Figure 4.19: An example of a single sensor, which counts each pedestrian who passes the counting zone (green) in either direction. The sensor stores the data onsite and regularly transmits the latest measurement to a server.
Source of image: https://youtu.be/isHEXkB2M4A?t=34

Using real-world data introduces a new set of challenges. Unlike the two previous scenarios, it is no longer possible to freely "craft" suitable system quantities or select the sampling rate. This means the underlying data characteristics can no longer be changed or averaged over multiple runs to reduce the noise. Furthermore, the Melbourne data is influenced by a myriad of non-recurrent, temporary and unknown exogenous factors which are not available in the data, such as city layout, weather, accidents, construction works, festivals or protests [Boukerche and Wang, 2020; Nagy and Simon, 2018; Vlahogianni et al., 2014].

In the Melbourne dataset instead of creating system states, I now describe how I "filter" the available dataset to obtain suitable data for the system identification and analysis. The first sensor record is in 2009, starting with 18 sensors and increasing to 71 sensors by the end of October 2021. Not only does the number of sensors vary, but sensors also vary in the degree to which they provide reliable data.

I select 11 sensors from the Melbourne dataset, taking data from a four-year interval between 2016 and 2019. Fig. 4.20 displays the selected sensors on a map of Melbourne (left) and outlines the time series collection data (right). Time intervals for which data is missing are empty. The first samples of each separate time series are marked in black. This becomes relevant for the time delay embedding performed during the modeling in the next section. Even though the available dataset includes more recent years, I do not include the years 2020 and 2021 in the analysis, because of an expected strong concept drift in the traffic patterns due to the SARS-CoV-2 pandemic[3].

[3]Melbourne was one of the world's most locked-down cities during the time of my thesis.

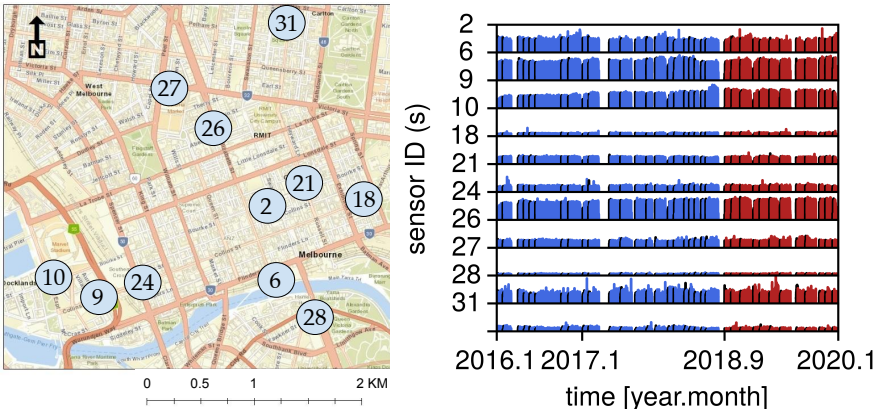


Figure 4.20: Left: Map of Melbourne, Australia, with corresponding sensor locations. Right: Schematic overview of data selection in Melbourne pedestrian count dataset. The training data in blue contains all states before September 2018 and the separated test data in red is used for the model analysis. The black-colored samples are required to perform a time delay embedding. The displayed sensor IDs match the original dataset. Adapted from Lehmberg et al. [2021].

In contrast to the bus scenario which shows mainly transient dynamics (cf. Fig. 4.13), the state evolution in Fig. 4.20 has reoccurring patterns within the four years. Despite missing exogenous factors in the sensor measurements (e.g. weather or events), traffic prediction is made possible through these reoccurring patterns at different time scales (e.g. daily, weekly or yearly) [Vlahogianni et al., 2014]. I assume that the underlying dynamical system is *ergodic*, that is, each state could be revisited if the system is observed for a long time.

For the system identification detailed in the next section, I build a single model to extract and forecast the traffic patterns in a multi-sensor setting. The advantage is that the forecasting quality of *all* sensors can improve from the cross-correlations. Ultimately, the intrinsic model components then provide a common basis of the (hidden) data-generating system on a city scale (Fig. 4.20, left). This differs to forecasting a single sensor only, for which the model adapts to patterns of the local sensor position. Furthermore, the multi-sensor state includes cross-correlations of multiple traffic patterns. A downside, however, is that a valid traffic state requires a valid measurement of *all* sensors. If a single sensor has an invalid measurement, the entire traffic state becomes invalid and is omitted in the dataset.

For the modeling of the data I also highlight the essential functionality of *datafold* to store *collections* of time series in the data structure `TSCDataFrame` (Section 3.3.1.1). Nevertheless, for an easier model analysis, I select the 11 sensors according to their reliability, to avoid highly fragmented time series. To fill small gaps in the sensor measurements of one or two hours, I insert the last observed valid measurement. Moreover, I remove short time series of fewer than two weeks to have larger batches of coherent time series. Finally, because I extract the intrinsic state dynamics without including ad-

ditional event information, I remove public holidays for the state of Victoria that fall on a weekday as "obvious events" that alter the traffic dynamics.

I store the final data selection of the Melbourne data in `TSCDataFrame`:

```
X_train: TSCDataFrame # two thirds of data (blue in Fig. 4.21)
X_test: TSCDataFrame  # one third of data (red in Fig. 4.21)
```

Each data structure separates the time series by gaps of missing data (Fig. 3.5 on 77). To follow the good practice in data-driven modeling, I split the available sensor data into a training and test set. As per Fig. 4.20 the split acknowledges the chronological order in the dataset, where older data are used to fit the model (blue) and more recent data for testing (red). Each of the two time series collections has the following form:

$$X_{\text{train/test}} = \left[\mathbf{x}_1^{(1)}, \ldots, \mathbf{x}_{J_1}^{(1)} | \ldots | \mathbf{x}_1^{(I)}, \ldots, \mathbf{x}_{J_I}^{(I)}\right] = \left[X^{(1)}, \ldots, X^{(I)}\right]. \tag{4.22}$$

All time series $X^{(i)} = [\mathbf{x}_1^{(i)}, \ldots, \mathbf{x}_{J_i}^{(i)}] \in \mathbb{R}^{[11 \times J_i]}$ have a different length (J_i) and maintain a constant sampling rate of $\Delta t = 1$ hour. The number of samples and time series for the training and test set is given in Table 4.2.

Table 4.2: Allocation of whole dataset into time series collections for training and test; visualized in Fig. 4.20 (right).

	nr. time series (I)	**nr. samples** $\left(\sum_i^I J_i\right)$
training	22	19,130
test	12	9,840
sum	34	28,970

About one third of the data is used for testing.

4.3.3 Operator-informed model architecture and parametrization

I now adopt the operator-based model architecture for the Melbourne sensor time series. The model architecture is suitable to achieve the main goal of my thesis: to extract geometric and dynamic coordinates by approximating the Laplace-Beltrami and Koopman operators from the time series data. I make use of the established numerical frameworks: (1) time delay embedding $\mathbf{g}(\text{td})$ (Section 3.4.2), (2) Diffusion Maps (DMAP) $\mathbf{g}(\text{dmap})$ (Section 3.4.3) and (3) Extended Dynamic Mode Decomposition (EDMD) (Section 3.4.4). While the first two points are responsible to project the time series data into a new state representation, the EDMD performs the Melbourne pedestrian traffic system. As highlighted in Section 2.4.4, all three methodologies have strong theoretical links. Importantly, I consider that the common assumption of an ergodic dynamical system is fulfilled for the Melbourne data.

For a detailed description of this setting, I refer to Section 3.1, from where the Fig. 4.21 is adapted to provide orientation for the Melbourne analysis. The main focus is to describe the final model parametrization used for the analysis.

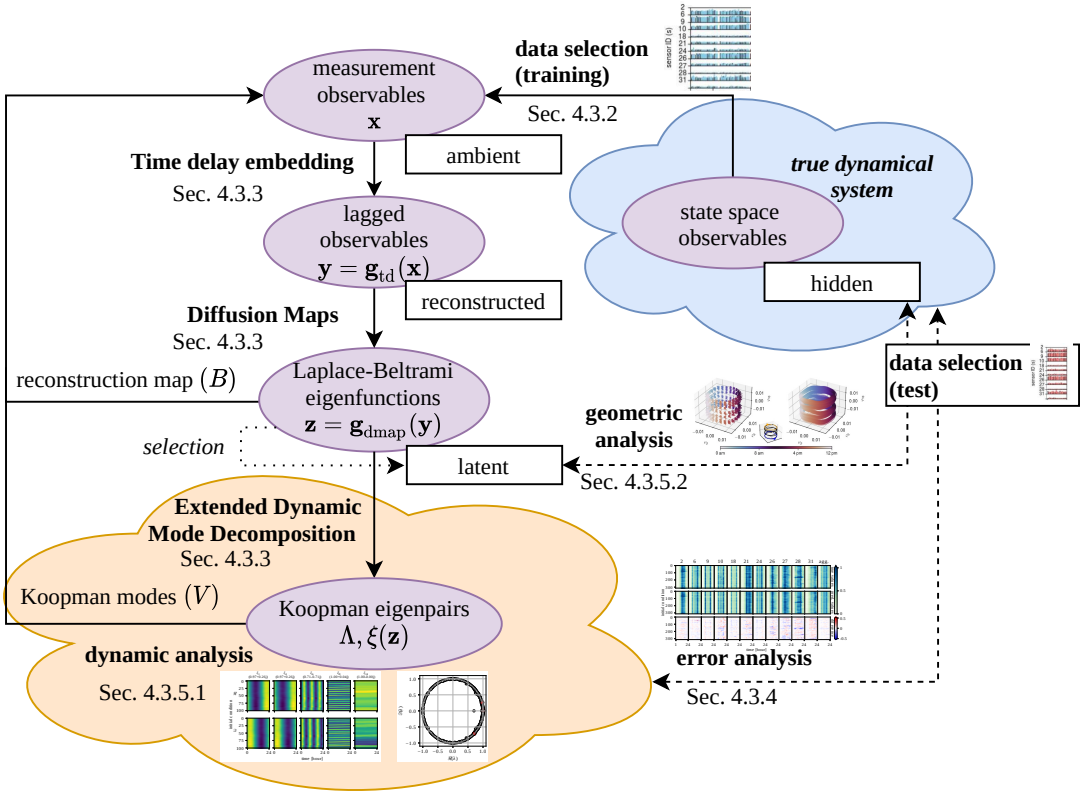


Figure 4.21: The main modeling setting in this thesis as described in Section 2.3. The images highlight the analyses performed in the next sections.

4.3.3.1 EDMD dictionary

The dictionary comprises the composed data transformation of time delay embedding $\mathbf{g}_{\text{td}}(\mathbf{x})$ on measurements states $\mathbf{x}$ and DMAP $\mathbf{g}_{\text{dmap}}(\mathbf{y})$ on time-delayed states $\mathbf{y}$. The transformation can be interpreted as a spatio-temporal feature extraction, where first temporal quantities are augmented and then sorted by relevance and reduced. Mathematically, the functions is represented as:

$$\mathbf{g}_{\text{td}}(\mathbf{x}_j; d, \kappa) = [\mathbf{x}_j, e^{-\kappa}\mathbf{x}_{j-1}, e^{-2\kappa}\mathbf{x}_{j-2}, \ldots, e^{-d\kappa}\mathbf{x}_{j-d}] = \mathbf{y}_j \in \mathbb{R}^{N(d+1)} \tag{4.23}$$

$$\mathbf{g}(\mathbf{x}) := \mathbf{g}_{\text{dmap}}(\mathbf{y}_j; \alpha, \mathcal{K}, P) = [\psi_1(\mathbf{y}_j); \ldots; \psi_P(\mathbf{y}_j)] = \mathbf{z}_j \in \mathbb{R}^P, \tag{4.24}$$

where $\mathbf{g}(\mathbf{x}) = \mathbf{z}$ describes the composed map from the original traffic states $\mathbf{x}$ to the data-intrinsic state representation $\mathbf{z}$. Later, the composed map is used in the EDMD

dictionary as a finite function basis to approximate the Koopman operator. However, the states themselves also have strong analytic power as they connect to the hidden state space of the identified dynamical system. I highlight this in the geometric analysis of Section 4.3.5.1.

The dictionary comes with a relatively small set of parameters in Eq. 4.23 and 4.24. Table 4.3 lists all six specified parameter settings for the Melbourne dataset. In the data analysis, I found that the concrete settings are robust; they produce similar results to those presented for the error analysis in Section 4.3.4 below. For this analysis, I varied different sensor configurations (including only using a single sensor). I therefore expect that the parameter setting would provide a good starting point when transferring the model architecture to similar traffic data, for example from other cities or also vehicular traffic [cf. Avila and Mezić, 2020]. For a clearer presentation of results, I only include the analysis of the 11 selected sensors in Fig. 4.20.

Because of the theoretical basis of each data transformation, I can either justify the parameter setting based on arguments or set up a heuristic. For deriving a heuristic I performed separate cross-validation runs on the Melbourne training data X_{train} by using the functionality of `EDMDCV` in *datafold* (Section 3.5.2). Because the model analysis in the later sections is performed on the completely separate test data X_{test}, the analysis is therefore unbiased from this procedure. Since the workflow is similar to the cross-validations already performed for the pendulum (Section 4.1.4) and bus station (Section 4.2.3) scenarios, I do not repeat this for the Melbourne dataset and instead focus on gaining insight into the final identified system.

Table 4.3: Parameters specified in the EDMD dictionary for the Melbourne sensor data.

Symbol	Setting	Description
d	168	The number of delays to be included in Eq. 4.23. Given the data interval of $\Delta t = 1$ this corresponds to a state that is embedded with the previous week of measurements.
κ	0	An exponential weighing factor in time delay embedding as a regularization parameter, as described in Berry et al. [2013].
α	1	The re-normalization factor in DMAP (Eq. 2.38 on page 45).
$\mathcal{K}(\mathbf{y}_i, \mathbf{y}_j)$	$\exp\left(-\frac{\|\|\mathbf{y}_j - \mathbf{y}_i\|\|_2^2}{2\varepsilon}\right)$	Default Gaussian kernel in DMAP as a geometric prior to describe the point similarity $\mathbf{g}_2$ (default).
ε	$\text{median}(\|\|\mathbf{y}_j - \mathbf{y}_i\|\|_2^2)$	The bandwidth of the Gaussian kernel to describe pairwise neighborhood relations.
P	500	The number of DMAP coordinates to compute. This also equals the size of the EDMD dictionary.

Time delay embedding is often essential for modeling traffic data; see Avila and Mezić [2020] and Cheng et al. [2021] for Koopman operator-based studies. The temporal feature extraction is needed because the collected measurements have typically no well-defined dynamics. A traffic state provides a good spatial description if many sensors are available but misses important temporal system information for the daily, weekly, or seasonal patterns within the system. A single instantaneous traffic state can be either part of a positive or negative trend, such as increasing or decreasing rush hour traffic on a weekday.

The final setting outlined in Table 4.3 highlights that I augment each traffic state with the states of the prior week. For smaller values of d, such as only embedding the last day ($d = 24$), I could observe that the model is unable to capture characteristic traffic changes between weekdays and weekends. Intuitively, this is because Friday has insufficient dynamic information to transition to the different traffic patterns of Saturday (the same argument applies to the change from Sunday to Monday). In a geometric perspective there are still "false neighborhood" relations in the data as depicted in Fig. 2.8.

Research that uses the composed transformation of 4.24 also suggests to set larger numbers of delays to further linearize the dynamics [Berry et al., 2013; Das and Giannakis, 2019; Giannakis, 2019]. However, this conflicts with the practical aspect of forecasting the system: an initial condition requires a time series of $d + 1$ states to obtain an intrinsic initial state $\mathbf{z}$. Fig. 4.20 marks the first d samples of each time series in black. For these states, there is no corresponding dictionary state $\mathbf{z}$ available and they cannot be part of the training or test set. Ultimately, I find that the final selection as per Table 4.3 is a good compromise between the two conflicting aspects of linearizing the system dynamics and the applicability for forecasting.

The time delay embedding also often includes an additional weighting factor, which is promoted in Berry et al. [2013]. However, in the cross-validation runs, I could not observe any improvement in prediction errors with the additional parameter. This might be due to the relatively small number of delays. I therefore disable the weighing by setting κ to zero.

The next three of parameters of Table 4.3 refer to DMAP (see Eq. 2.37 – 2.44 on page 45). The method describes the second (nonlinear) map $\mathbf{g}_{\text{dmap}}(\mathbf{y})$, where the scalar functions $\psi_p(\mathbf{y})$ approximate the eigenfunctions of the Laplace-Beltrami operator. These provide the geometrically-aligned function basis as a central element of the architecture in Fig. 4.21. In short, I refer to the eigenfunctions also as "DMAP coordinates".

Within DMAP I set $\alpha = 1$ to perform an additional normalization of the point density in the embedded samples $\mathbf{y}$. This improves the convergence guarantees to obtain eigenfunctions of the Laplace-Beltrami operator [Coifman and Lafon, 2006a]. The extracted geometry from the point cloud is induced by the Gaussian kernel as a standard kernel to measure the point similarity in the data [Berry and Sauer, 2016]. The kernel itself introduces a bandwidth parameter ε, which I set to the median value of point-wise Euclidean distances in $\mathbf{y}$. This bandwidth choice is a result of the cross-validation that I performed on the training data.

The last parameter $P = 500$ in Table 4.3 specifies the number of DMAP coordinates $\{\psi_p\}_{p=1}^{P}$ to include in the intrinsic state representation. The parameter therefore specifies the "richness" of the function basis $\mathbf{g}(\mathbf{x})$. The functions are inherently ordered by their respective eigenvalue which relates to the smoothness of the function. Because of these properties, the parameter P corresponds to a truncation which can regularize a model that operates on these states (later the EDMD): functions $\psi_p(\mathbf{y})$ with a small eigenvalue are more likely to associate to noise-corrupted coordinates and can therefore be truncated [Giannakis, 2019]. I could observe that the error within the final forecasting setting (Section 4.3.4) decreases until $P = 500$. The corresponding eigenvalue of the last coordinates has an order of $\omega_{500} = \mathcal{O}(10^{-5})$.

4.3.3.2 EDMD model

The final EDMD model requires only the dictionary $\mathbf{g}(\mathbf{x})$ of Eq. 4.24 (with parametrization in Table 4.3) and the training time series collection X_{train} from Eq. 4.22. I now summarize how to specify and build the actual `EDMD` model in *datafold* to capture the full data processing pipeline of Fig. 4.21:

```
X_train: TSCDataFrame # training data from Eq. 4.21

# initialize time delay embedding
takens = ("takens",TSCTakensEmbedding(delays=168))

# initialize the Diffusion Maps algorithm
# (D is the pairwise Euclidean distance matrix)
median_epsilon = lambda D: np.median(D)
laplace = ("laplace",
              DiffusionMaps(
              kernel=GaussianKernel(epsilon=median_epsilon),
              n_eigenpairs=500,
              alpha=1,)
          )

# specify final model with a slightly modified EDMD class
edmd = EDMDPositiveSensors(
              dict_steps=[takens, laplace],
              dmd_model=DMDFull(is_diagonalize=True),
              include_id_state=False,
              sort_koopman_triplets=True,
)

# build model
edmd.fit(X_train)
```

First, the EDMD dictionary is specified according to Eq. 4.23 – 4.24 and Table 4.3: `takens` ($\mathbf{g}_{\text{td}}(\mathbf{x})$) and `laplace` ($\mathbf{g}_{\text{dmap}}(\mathbf{y})$). Both transformations are chained in the EDMD dictionary of `EDMDPositiveSensors`. The class is a specific adaptation of the standard `EDMD` class from Section 3.5.1, where all (unrealistic) negative sensor forecasts are set to zero. Negative values can occur because the model does not include an explicit constraint.

Mathematically, the EDMD model (without the extra treatment of negative values) can be described in terms of the Koopman triplet in a matrix form (see derivation in Eq. 2.26 – 2.29):

$$\hat{\mathbf{x}}_t = V\Lambda^{t/\Delta t}\xi(\mathbf{z}_1), \tag{4.25}$$

where $\hat{\mathbf{x}}_t \in \mathbb{R}^{11}$ contains the traffic state prediction of the 11 sensors, $V \in \mathbb{C}^{[N\times P]}$ the modes and $\Lambda \in \mathbb{C}^{[P\times P]}$ is a diagonal eigenvalue matrix. The (vectorized) function $\xi(\mathbf{z})$ can be interpreted as a spectrally-aligned state of the linear system in Eq. 4.25:

$$\xi(\mathbf{z}_1) = [\xi_1(\mathbf{z}_1), \ldots, \xi_P(\mathbf{x}_1)] = \Phi^{-1}\mathbf{g}(\mathbf{z}_1) = \Phi^{-1}\mathbf{z}_1, \tag{4.26}$$

where $\xi_p(\mathbf{z})$ is a Koopman eigenfunction, $\mathbf{z} \in \mathbb{R}^P$ the intrinsic state, $\mathbf{g}(\mathbf{x})$ the dictionary, and Φ^{-1} the left eigenvectors of the Koopman matrix $\mathcal{U}_{\Delta t} = \Phi\Lambda\Phi^{-1}$ (cf. Eq. 2.29). Note that obtaining the initial state $\mathbf{z}$ requires a time series of length $d + 1 = 169$ (i.e. the minimum number to perform the time delay embedding).

All three Koopman components are available through the `EDMDPositiveSensors` class after the model is constructed. In the next section, I analyze the prediction accuracy of the EDMD model and continue in Section 4.3.5 to analyze the internal model components to gain insight into the identified dynamical system.

4.3.4 Error analysis of day ahead forecasting

The main purpose of this section is to show that the EDMD model can accurately reconstruct the sensor measurements in the training data and generalize to the separate test data (as depicted in Fig. 4.20). The error analysis occurs in a multi-sensor setting, where all 11 sensors in the data are simultaneously predicted for 24 hours. I intentionally use a relatively large test set (about $1/3$ of the data) to include a large variety of initial conditions for the analysis. In a more application-oriented setting where the main objective is to minimize prediction errors, it is common to frequently re-train the model over time to integrate more recent data. This can be realized in a sliding window scheme or by adapting the methods to a streaming setting; for DMD see Hemati et al. [2014].

To promote a structured analysis of the model's forecasting performance I allocate both the training and test data in *pairs* of time series

$$\left\{\left(X_1^{(c)}, X_{\text{true}}^{(c)}\right)\right\}_{c=1}^{C_{\text{train/test}}}. \tag{4.27}$$

Fig. 4.22 shows an example of the structured forecasting setting. The number of pairs in the training set is $C_{\text{train}} = 622$ and for the test set $C_{\text{test}} = 314$. The left time series

$X_1^{(c)}$ in the tuple serves as the initial condition and is the only information available for a model to perform a 24 hour forecast. It includes the entire past week plus the most recent state, $X_1^{(c)} = [\mathbf{x}_{-168}^{(c)}, \dots, \mathbf{x}_0^{(c)}] \in \mathbb{R}^{[N \times 169]}$. The last state $\mathbf{x}_0^{(c)}$ always corresponds to midnight of a day and is the reference state for the initial condition; see markers in Fig. 4.22. The right time series in Eq. 4.27 contains the measured sensor data of the subsequent 24 hours of the respective day, $X_{\text{true}}^{(c)} = [\mathbf{x}_1^{(c)}, \dots, \mathbf{x}_{24}^{(c)}]$. All time series to be predicted in $X_{\text{true}}^{(c)}$ are disjoint to each other, whereas the initial condition time series $X_1^{(c)}$ overlap.

Using a pairs of initial condition and subsequent 24 measurements, it is now possible to initialize and forecast a respective time series $X_{\text{pred}}^{(c)}$ by evaluating the EDMD model in Eq. 4.25:

$$\hat{\mathbf{x}}_j^{(c)} = V \Lambda_{\Delta t}^j \xi(\mathbf{z}_1^{(c)}), \text{ for } j = (1, \dots, 24), c = (1, \dots, C_{\text{train/test}}) \tag{4.28}$$

$$\text{such that also } [\hat{\mathbf{x}}_j^{(c)}]_k = \max\left\{[\mathbf{x}_j^{(c)}]_k, 0\right\} \text{ for } k = (1, \dots, 11). \tag{4.29}$$

The $\hat{\mathbf{x}}_j^{(c)}$ is the model prediction for a corresponding true measurement $\mathbf{x}_j^{(c)} \in X_{\text{true}}^{(c)}$. The Koopman eigenfunction is evaluated according to Eq. 4.26, such that the dictionary maps the reference state of the time series $X_1^{(c)}$ to the initial state $\mathbf{g}(\mathbf{x}_0^c) = \mathbf{z}_1^{(c)}$. The second statement describes the additional model adaptation of setting negative forecasts in the k-th sensor to zero.

With the built EDMD model and the data allocation of the prediction setting, the model can be evaluated by using the prediction API of *datafold*:

```
# initial condition time series of length 169
# (left in Eq. 4.21)
X_ic: TSCDataFrame

# test time series to be predicted (right in Eq. 4.26)
X_test_windowed: TSCDataFrame

# 24 hour prediction for each initial condition
X_pred = edmd.predict(X_ic, np.arange(1, 25))

# evaluate difference in model
difference = X_pred - X_test_windowed
```

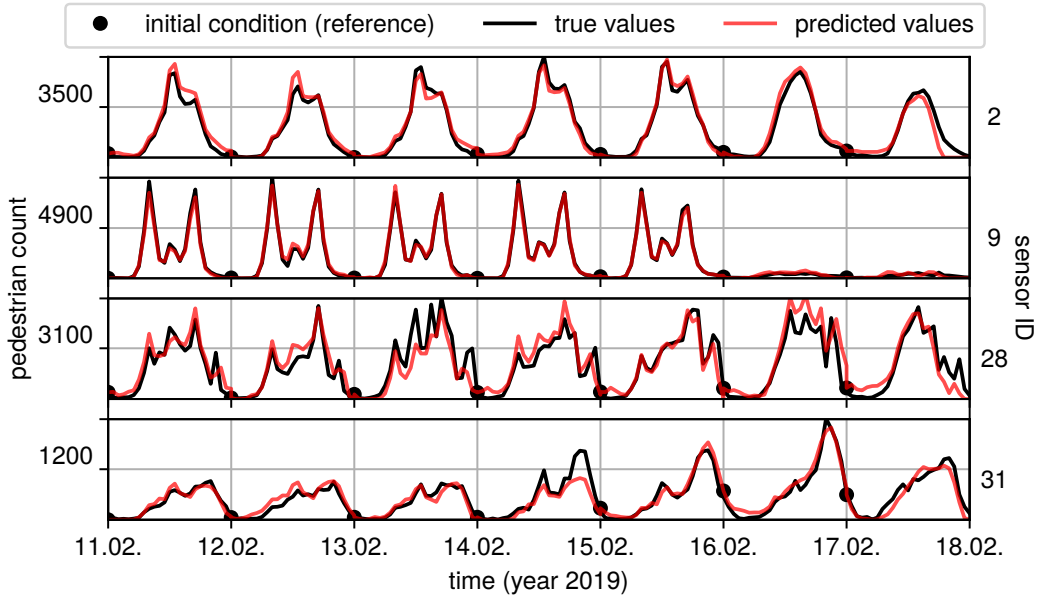


Figure 4.22: A representative week (starting on Monday) of four of the 11 sensors, with distinctive traffic patterns. Each sensor's plot is scaled to the maximum observed pedestrian count of the respective sensor in this week. The initial conditions (black markers) are the reference states at midnight ($\mathbf{x}_0^{(c)}$). The true (black) and estimated (red) sensor measurements of the following 24 hours are compared in the error analysis. Note that each state of a time series includes all sensor measurements simultaneously. Adapted from Lehmberg et al. [2021].

For a comparison of the model's error performance, I also set up a heuristic baseline model, which naively forecasts the future time series based on the sensor measurements of the same day of the previous week:

$$\hat{\mathbf{x}}_j^{(c)} = \mathbf{x}_{j-168}^{(c)}, \text{ for } j = (1, \ldots, 24). \tag{4.30}$$

For the error analysis I first look at the mean and standard deviation of the difference values between true and predicted time series ($X_{\text{true}} - X_{\text{pred}}$). Table 4.4 includes an overview of the error statistics per sensor. As already mentioned, the collected measurement data can be expected to be noisy due to missing explanatory factors in the data. For this reason, a good model should only describe *unbiased* and meaningful (slow) dynamics of the system. High-frequency dynamics are increasingly noise-corrupted and should therefore be truncated by a model.

Table 4.4: The error statistics of the EDMD model for both the training and test set. The errors are listed per sensor and evaluated on the prediction pairs X_{true} and X_{pred} from Eq. 4.27. The sensor IDs match with the original data source in Melbourne [2021a]. The mean and standard deviation the difference values. The relative root mean squared error (RRMSE) is computed with Eq. 4.31. The prediction errors of the baseline model of Eq. 4.30 are in columns with suffix (b). The last row aggregates (agg.) the RRMSE to a mean error metric over all sensors. Adapted from Lehmberg et al. [2021].

	training ($C_{\text{train}} = 622$)				test ($C_{\text{test}} = 314$)		
ID (s)	$Q_{95\%}$	mean$\pm$std	RRMSE	RRMSE(b)	mean$\pm$std	RRMSE	RRMSE(b)
2	3116	-8 $\pm$ 198	6.36%	8.69%	-11 $\pm$ 204	6.55%	7.06%
6	4048	-3 $\pm$ 199	4.91%	6.82%	1 $\pm$ 254	6.28%	8.03%
9	2805	-2 $\pm$ 103	3.68%	4.54%	-6 $\pm$ 146	5.20%	5.83%
10	592	0 $\pm$ 48	8.11%	10.34%	-4 $\pm$ 57	9.62%	11.46%
18	1540	-1 $\pm$ 53	3.47%	4.25%	6 $\pm$ 74	4.81%	5.50%
21	1410	-2 $\pm$ 118	8.40%	11.82%	-9 $\pm$ 130	9.27%	11.51%
24	3970	-2 $\pm$ 138	3.48%	4.65%	8 $\pm$ 170	4.29%	5.11%
26	1629	-1 $\pm$ 109	6.69%	8.24%	4 $\pm$ 168	10.29%	12.08%
27	339	0 $\pm$ 30	8.98%	11.64%	4 $\pm$ 35	10.30%	11.88%
28	2662	-4 $\pm$ 293	11.01%	16.25%	1 $\pm$ 350	13.16%	17.99%
31	719	-2 $\pm$ 81	11.33%	14.57%	-10 $\pm$ 95	13.26%	15.70%
agg.		-3 $\pm$ 171	6.95%	9.26%	-1 $\pm$ 207	8.46%	10.20%

According to the Table 4.4, all sensor difference values have a mean close to zero, both for the training and test set. I therefore conclude that on average the model is unbiased, despite unobserved events in the data. The standard deviations between sensors are a way to quantify how well a sensor can be predicted (based on past measurements). For example, sensor 9 has much "cleaner" patterns and lower standard deviation, when compared to sensor 28. This becomes apparent in the example week in Fig. 4.22.

Compared to the training set, both the mean error and standard deviation increases in the test set. One obvious reason is that the training data are merely a reconstruction of the data that are used to build the model, while the test data contains out-of-sample initial conditions. Another factor that poses a general challenge for time series modeling is that the chronologically separated test data may contain concept drifts that are not included in the training data. These can be temporary short changes, such as construction works, but also long term trends. For example, an average annual growth rate of 3.68% for the population of the City of Melbourne for parts of the considered time interval [Carter et al., 2020, Fig. 4]. As a result, the forecasting errors of the time-invariant EDMD model can vary or increase over time if the model diverges from the true system. While this seems to contradict my initial assumption that the Melbourne traffic system is ergodic, I argue that these long term trends can be neglected, as the dominating patterns are on a daily and weekly basis, which I highlight below.

In Table 4.4 I also include the relative root mean squared error (RRMSE) of both the EDMD model and the baseline model from Eq. 4.30. The metric corresponds to a standard RMSE relative to the 95% quantile of each sensor's measurement values computed on the training data; see column $Q_{95\%}$ in the table. For a single sensor s with (scalar) measurements the metric computes with

$$\text{RRMSE}(s) = \frac{1}{Q_{95\%}^{(s)}} \sqrt{\frac{\sum_{m=1}^{M} \left(\hat{x}_m^{(s)} - x_m^{(s)}\right)^2}{M}}, \tag{4.31}$$

where $x^{(s)}$ is the measurement or forecast (with a hat) of the sensor. The sum indexes over all $m = (1, \ldots, C_{\text{train/test}} \cdot 24 = M)$ measurements of the training and test set in Eq. 4.27. Because of the squared sum, RRMSE penalizes greater difference values between the model and forecast.

The comparison of the EDMD model with the baseline of Eq. 4.30 shows that the EDMD improves the predictions for all individual sensors and the aggregated metric in the last row with 2.31% for training and 1.74% for the test data. This is not by a large margin, however, indicating that the baseline is already a good estimator due to the periodicity in traffic. It is therefore reassuring that the EDMD model captures these principal dynamics and further improves them. More importantly, the EDMD model now includes data-specific patterns, which provide insight into the model and underlying system. This is not possible with the baseline model, nor with many methods that promote complex structures to model capture nonlinear and high-dimensional dynamics.

I also compare error results to other available studies that analyze and predict the sensors of the Melbourne dataset. However, a direct comparison is difficult because the studies often lack detailed error rates, specific sensor locations (IDs) or information on the exact forecasting setting. Karunaratne et al. [2017] investigate the Melbourne data in a 24 hour prediction setting for 10 (unspecified) sensors. They use a Gaussian process in which they specify a "working-week kernel". For the error analysis they specify a mean error relative to $\bar{\mathbf{x}}$ (MER), where $\bar{\mathbf{x}}$ is the mean sensor observation of the respective day of the prediction. The error is evaluated on a single test month (October 2016), for which they report an MER between 16.53 and 20.68 for three proposed model variations. Furthermore, they also include an ARIMA baseline model with an MER of 40.11. Compared to the EDMD model in my setting, I could evaluate an MER of 16.98 for the selected 11 sensors. While this is not far away from the best result reported in Karunaratne et al. [2017], I test the model on a significantly larger test set of 314 days (compared to 30 days in the other study).

Wang et al. [2018] predict the Melbourne sensor measurements for time horizons between 24 and 192 hours. In a model selection procedure, the authors present an ARIMA model that performs best against the alternative models SVR and multiple linear regression. However, the error rates are only presented in a figure. Moreover, the selected error metric — the mean absolute percentage error (MAPE) — is in my

opinion, not a suitable metric: if a sensor measurement has a pedestrian count of zero, this leads to a division by zero.

Proceeding with the forecasting analysis of the EDMD model, I highlight that the model can adequately capture the distinct sensor patterns. Fig. 4.23 visualizes the true (first row) and predicted sensor values (second row). The figure only includes the test data, which is aligned to the 24 hour prediction setting: Each (color-coded) row in a sensor's block corresponds to the prediction based on an initial condition $c = 1, \ldots, C_{\text{test}}$ and has 24 entries, $j = 1, \ldots, 24$, corresponding to the hours of a day.

Most of the time adjacent rows are in a calendrical sequence. However, there are also intervals of missing data in the test data, which can lead to interruptions of the patterns in a vertical direction. Because the sensors have different scales in terms of usual pedestrian volume all values are scaled by the computed 95% quantiles listed in Table 4.4. The bottom row in Fig. 4.23 shows a heatmap of the difference values between the scaled true sensor measurements and model estimations.

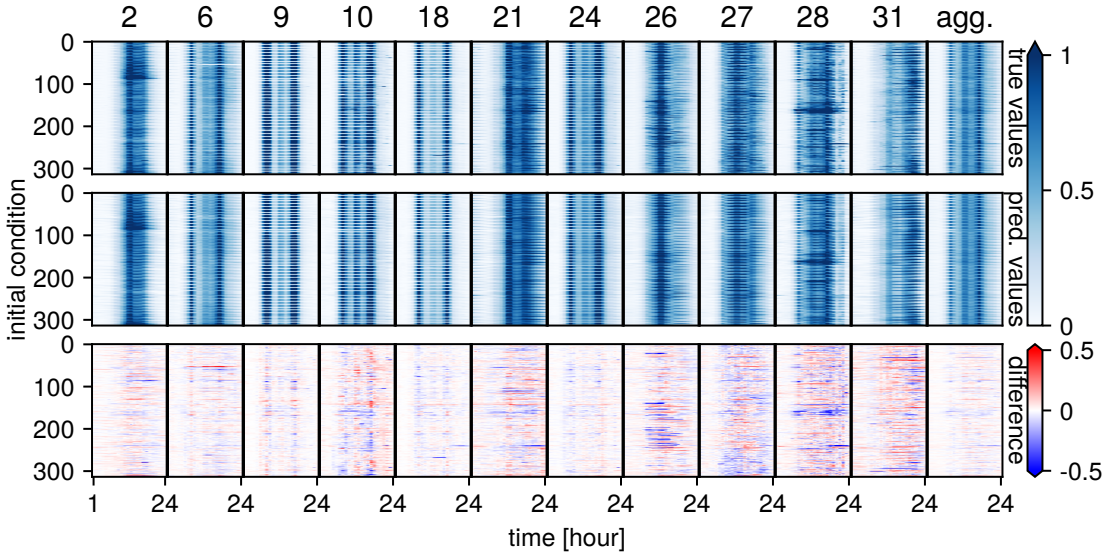


Figure 4.23: Each column corresponds to the sensor ID at the top. The true sensor measurements (first row) are compared to the model estimations (second row) of the test set. The third row displays the difference, $X_{\text{pred}} - X_{\text{true}}$, indicating over-estimation (red) or under-estimation (blue). The values are scaled by the computed quantiles $Q_{95\%}$ in Table 4.4. Adapted from Lehmberg et al. [2021].

The sensor profiles in Fig. 4.23 highlight distinctive traffic patterns at each sensor. This is complemented by Fig. 4.22, showing an example week of four representative sensors. Sensor 2 has typically a single peak on weekdays and only moderately decreased traffic on weekends. Sensors 9 and 18 show a typical rush hour pattern with two peaks in pedestrian volume in the morning and evening. Moreover, the pedestrian traffic is much lower on weekends, resulting in horizontal stripes in the two sensors' profiles in Fig. 4.23.

In contrast to the previous patterns, sensor 28 has typically high fluctuations during a day with multiple peaks. These can be observed both during the week and on the weekend. Sensor 31 is different to other sensors in that it has increased traffic on the weekend and higher relative traffic during the night.

There are also clear changes in traffic patterns on a seasonal time scale. The darker colored regions in sensor 2 before initial conditions 100 and past 300 are days in December. Because sensor 2 is located at a shopping precinct this shows the effect of Christmas sales. The seasonal patterns are regular and reoccurring and the model can adapt to such temporary concept drifts to some extent. However, the adaptation to a (temporary) change usually results in larger errors, because information only becomes available to the model in the "initial condition window" $X_1^{(c)}$ (cf. Eq. 4.27).

Often the regular traffic is also "disturbed" by irregular and unpredictable "random events" (e.g. construction works or extreme weather). These are often intermediate concept drifts and usually result in larger error bursts in the model forecast. A high discrepancy between forecast and measured traffic can be a sign of traffic anomalies, which for the Melbourne dataset is analyzed in Doan et al. [2015]; Zameni et al. [2019]. An example of a short term increase in traffic volume can be seen at sensor 28 between prediction 100 and 200.

Based on the color-coded sensor profiles in Fig. 4.23, I conclude that the EDMD model is able to capture weekly, daily and to some extent seasonal dynamics. All patterns of the sensors could be forecasted simultaneously and are captured in the model. This means that sensors with a large pedestrian volume are not favored over ones with less traffic.

Notably, the model structure encodes all patterns in the intrinsic state representations that relate to the Laplace-Beltrami operator and the spectral components of the Koopman operator. In the next section, I look into these model objects in more detail.

4.3.5 Gaining insight into the identified system

I now go beyond the typical error analysis in a data-driven modeling workflow. I investigate the intrinsic pedestrian patterns extracted from the Melbourne dataset and stored as internal "building blocks" within the EDMD model. All 11 sensors are all located in the City of Melbourne, from which I assume that the time series collection stems from a single data-generating system, evolving the pedestrian traffic on a city scale (denoted as "true dynamical system" in Fig. 4.21).

As highlighted in the previous section all distinctive spatio-temporal sensor patterns are encoded in a single EDMD model. In Section 4.3.5.1, I use the intrinsic state representation of the approximated eigenfunctions of the Laplace-Beltrami operator in Eq. 4.24 for a *geometric* analysis. In Section 4.3.5.2, I show how the spectral components of the Koopman operator in Eq. 4.25 give insight to the *dynamics* of the identified system.

4.3.5.1 Geometric state space analysis with Laplace-Beltrami operator

The first data transformation on the original sensor measurements to obtain an intrinsic state representation is the time delay embedding $\mathbf{g}_{\text{td}}(\mathbf{x}) = \mathbf{y}$ from Eq. 4.23. As highlighted in Section 3.4.2 (cf. Fig. 2.8 on page 48), this induces a new geometry on the augmented sensor data. According to Takens theorem, if sufficiently many delays are included this reconstructs a qualitative equivalent manifold to the state space of the attractor for an ergodic dynamical system [Deyle and Sugihara, 2011]. However, the delayed states are high-dimensional and include many highly correlated quantities in $\mathbf{y} \in \mathbb{R}^{1859}$ (from $N \cdot (d+1)$ with $N = 11$ and $d = 168$ delays), due to temporal neighborhood relations.

I therefore assume that the manifold assumption is fulfilled and the new geometry is actually much lower dimensional than the embedded state. This justifies the second map in the composed transformation where DMAP performs a nonlinear map onto the final state representation $[\psi_1(\mathbf{y}), \ldots, \psi_P(\mathbf{y})]$, as per Eq. 4.24. The function coordinates $\psi_p(\mathbf{y})$ correspond to the eigenfunctions of the Laplace-Beltrami operator (I also refer to the functions also as "DMAP coordinates").

While the functions serve as a finite basis to approximate the Koopman operator in the EDMD dictionary, they also contain valuable geometric system information, with a strong connection to manifold learning (see Section 2.4.2). In particular, research shows that the composed transformation has strong time scale separation qualities, in which slow dynamics can be extracted, similar to a Fourier analysis [Belkin et al., 2009; Berry et al., 2013]. While in the Melbourne dataset I computed 500 eigenfunctions, there is a parsimonious set of functions that describe principal geometric directions of the inferred manifold [Dsilva et al., 2018]. For the analysis, I mainly rely on a visual representation of the intrinsic patterns, provided by the DMAP coordinates. However, it is also possible to apply further numerical procedures, such as estimating the manifold dimension [Strange and Zwiggelaar, 2014] or automatically identifying principal geometric coordinates [Dsilva et al., 2018].

With the EDMD model and *datafold* specifications, the coordinates are easily accessible:

```
X_test: TSCDataFrame  # test data

# compute intrinsic state representation
Psi = edmd.transform(X_test)

# example to access second coordinate
dmap_coordinates.iloc[:, "psi2"]
```

The variable `Psi` is again a time series collection (of type `TSCDataFrame`) that contains the function evaluations $\psi_p(\mathbf{y}_j^{(i)}) = [\mathbf{z}_j^{(i)}]_p$ (similar to Eq. 4.22 in the intrinsic states). Since `X_test` is not part of the training data, the coordinates are mapped with the Nyström extension in DMAP (cf. Eq. 2.43 – 2.44 on page 45).

Fig. 4.24 displays 10 selected DMAP coordinates $\psi_p(\mathbf{y})$. The coordinate values are visually aligned in the 24 hour prediction setting, which I established in the previous section (cf. Eq. 4.27 and Fig. 4.24). The images correspond to a two-dimensional projection of the scalar eigenfunctions ψ. In the figure I selected the eigenfunctions based on "interesting patterns" that are apparent in the visual profile. In each image I only display the first 100 initial conditions of the test data, because the patterns mostly repeat along the initial condition axis.

The most dominant direction on the data-inferred manifold is the first non-trivial eigenfunction, ψ_2, after the constant ψ_1. The visual profile shows a simple sinusoidal function over the 24 hour prediction horizon and hardly depends on the initial condition. The next function, ψ_3, is very similar in that it is also sinusoidal with a shifted phase. Only looking at these first two functions, I deduce that the two coordinates (ψ_2, ψ_3) describe a circle. As such the coordinates acknowledge the periodicity in a day, which is further strengthened in the data projection of Fig. 4.26.

The eigenfunctions to follow, ψ_4 to ψ_9, basically correspond to harmonic functions with higher frequencies in a day. In the data projection, they describe the same direction. While in the finite function basis to approximate the Koopman operator these functions enrich the function basis, in the geometric state space analysis these "repeated eigenfunctions" do not contain new essential information. This is because the coordinates align in the direction on the manifold which is already covered in previous coordinates [Dsilva et al., 2018] — in this case intraday features. To obtain a parsimonious description of the inferred state space geometry, it is essential to factor out the repeated eigenfunctions. Ultimately, this leads to coordinates that describe slow (macroscale) dynamics of the system [Dsilva et al., 2018].

In the data projections of Fig. 4.24 a new pattern appears in ψ_{10}. The function is again sinusoidal but now varies along the axis of the initial conditions. Notably, the lines formed by the peaks (or troughs) are slightly tilted and therefore slowly change over time. This means that in the 24 hour prediction, the range of values in ψ_{10} is narrow. This differs from the previous functions, where the full image of the function appears in a single day. This "slower" change in coordinates over time makes it possible to distinguish states on larger time scales. For ψ_{10} this corresponds to the days in a week (Fig. 4.26). As with the previous intraday direction, coordinates of higher oscillations also appear, which can be identified as repeated directions (e.g. ψ_{12} and ψ_{18}). Since the coordinates are ordered by the DMAP eigenvalues, I infer that the intraday features are most important on the reconstructed state space geometry, followed by weekly features.

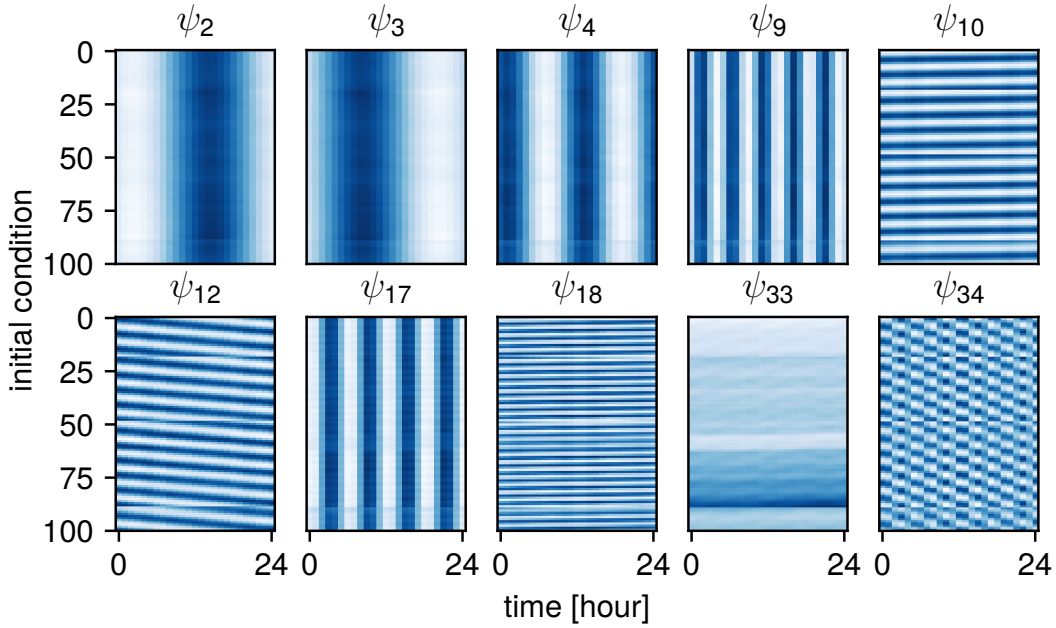


Figure 4.24: A selection of DMAP coordinates based on visual features. The values are part of the dictionary state, which comprises all 500 eigenfunctions in the EDMD dictionary. Only the first 100 time series of the test set are visualized in the 24 hour prediction setting. Adapted from Lehmberg et al. [2021].

Fig. 4.24 also includes coordinates that combine previous principal eigendirections. An example is given with coordinate ψ_{34}, where the two leading directions (intraday and weekly) form a checkered pattern. Again, such coordinates and higher oscillations thereof do not carry new geometrical information but are nevertheless suitable for inclusion in the function basis as the EDMD dictionary.

The last function with an interesting pattern is ψ_{33}. It differs from the other visual profiles by not showing an obvious periodic pattern in the data alignment. However, in Fig. 4.25 I expand the coordinate to the entire time horizon of the selected Melbourne data (both training and test data). As previously mentioned, sensor 2 is a good example to identify the seasonal pattern of Christmas sales, as indicated by increased traffic in December. In Fig. 4.25 it is apparent that the peaks of the intrinsic coordinate ψ_{33} match with the increased traffic during December. Notably, temporary high traffic during the year does not "disturb" the intrinsic coordinate. This is also because ψ_{33} is a coordinate of *all* sensor measurements, which can prevent "false positives". Moreover, the peaks of ψ_{33} roughly correlate to the traffic volume at each year. This shows two interesting aspects: (1) the model can extract seasonal patterns from the data and (2) the importance of this seasonal "Christmas pattern" is highlighted in coordinate index 33.

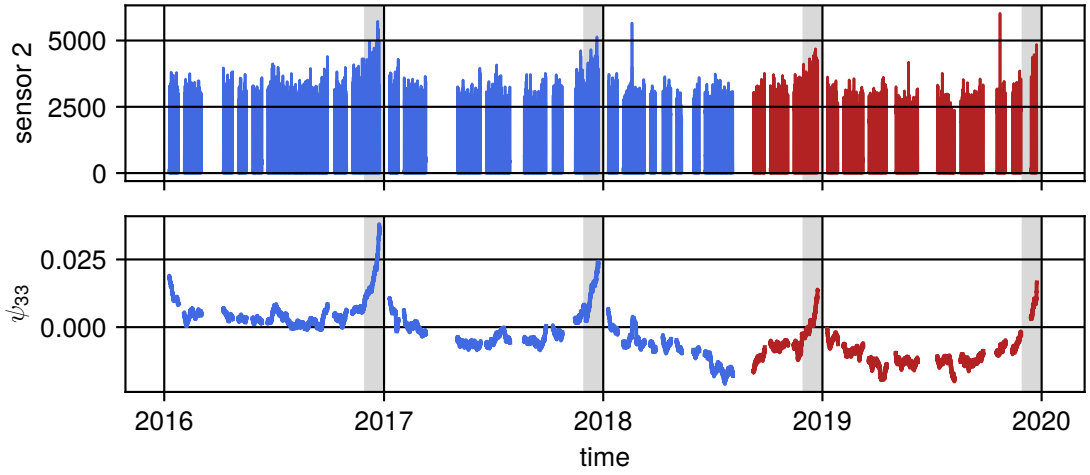


Figure 4.25: The seasonal "Christmas pattern" in the measurements of sensor 2, compared to the intrinsic coordinate ψ_{33}. The data includes both the full dataset with the training in blue and test set in red. The month of December is highlighted in gray where the traffic increases due to increased shopping activity before Christmas.

In Fig. 4.26, I can now select the principal DMAP coordinates and visualize the test data in a projection of the identified state space manifold. Here I use the three geometrically dominant functions $(\psi_2, \psi_3, \psi_{10})$. The left graph displays the original coordinates as evaluated for the test data in a point cloud and with a discrete sampling rate $\Delta t = 1$ hour. From the visual inspection, the "global point cloud" of the data subdivides into smaller point clouds. In the cyclic order in (ψ_2, ψ_3), the small point clouds are separated by the hour and partly separated in the vertical axis of ψ_{10}. Collectively, the point cloud forms a hollow cylinder-like geometry.

The small graph at the center gives a schematic sketch of an average week in the test data in terms of these coordinates. The colored parts at the bottom and top of the time series highlight that the coordinates divide the week into two equal parts. The top marker of the cylinder corresponds to Sunday 8 pm, from where the orbit declines to Wednesday 8 am and vice versa.

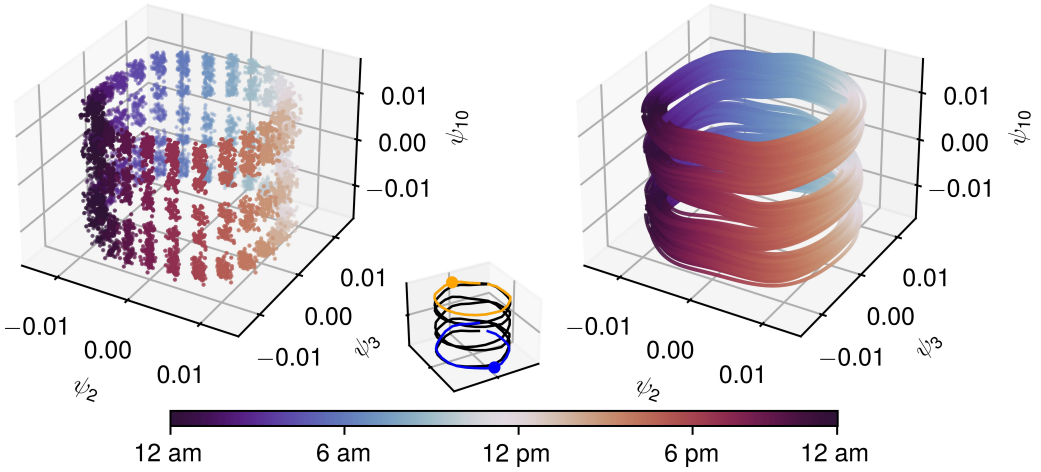


Figure 4.26: Projection of the intrinsic states $\mathbf{y}$ onto the geometrically leading coordinates $(\psi_2, \psi_3, \psi_{10})$. The cyclic color-code matches the time of the day for the respective point. Left: The evaluations of the intrinsic states at the prediction of the test data. Right: The interpolated time series along the flow using the Koopman matrix. Small center plot: A schematic flow of a week, as the average of all time series in the test data. The yellow colored part corresponds to Wednesday with a marker at 8 pm and the blue colored part to Sunday with a marker at 8 am. Adapted from Lehmberg et al. [2021].

However, the discrete point distribution in the left graph of Fig. 4.26 is merely an artifact of the discrete sampling in the data. While the "classical" manifold learning only investigates static data, I can also utilize the inferred dynamics in the EDMD model to interpolate between the intrinsic states. This only requires a small adaptation of the original model:

$$\mathbf{z}_t = \overbrace{\Phi \Lambda^{t/\Delta t} \Phi^{-1}}^{U_{\Delta t}^{t/\Delta t}} \mathbf{z}_1. \tag{4.32}$$

Instead of reconstructing the measurement states $\mathbf{x}$ with the Koopman modes V, the system remains in the intrinsic state representation (this follows from Eq. 2.30 – 2.32 on page 35). For details on the mathematical treatment of the interpolation, I refer the reader to the software treatment of DMD methods in *datafold*, Section 3.4.4.

In the right graph of Fig. 4.26, I use Eq. 4.32 to interpolate the three selected geometric coordinates by evaluating the linear system per minute instead of per hour as in the original data. With the state interpolation, I obtain a smooth description of the identified state space geometry (here projected onto the three leading coordinates). The visualized object suggests an intrinsic dimension of two. But because of the periodicity in (ψ_2, ψ_3) it cannot be further reduced. Moreover, because the time series cross in the vertical direction of ψ_{10}, this suggests that the intrinsic manifold dimension of the state space is larger than two. One possible way to further improve the visualization is to project the four leading (periodic) coordinates on a torus manifold.

4.3.5.2 Dynamics analysis with the Koopman operator

I continue the model analysis by investigating the three spectral components of the approximated Koopman operator: the modes $\mathbf{v}_p$, eigenvalues λ_p and eigenfunctions $\xi_p(\mathbf{z})$ which collectively describe a linear dynamical system,

$$\hat{\mathbf{x}}_t = \sum_{p=1}^{P=500} \mathbf{v}_p \lambda_p^{t/\Delta t} \xi_p\left(\mathbf{z}_1\right). \tag{4.33}$$

Despite being linear, the system captures the nonlinear traffic patterns of all sensors collectively, as highlighted in Fig. 4.23. The spectral system properties are independent of the observation modality. In other words, measurements from different sensors would yield the same result (assuming sufficiently rich information and removing noise effects) [Giannakis, 2019].

In the EDMD model, all objects are computed from a spectral decomposition of the Koopman matrix, which is obtained from a linear system identification in intrinsic state time series $\mathbf{z}_j$; Eq. 2.26 – 2.32). Within the software model, all of the components are accessible with:

```
X_test: TSCDataFrame

edmd.koopman_modes_
edmd.koopman_eigenvalues_
edmd.koopman_eigenfunctions_(X_test)
```

I first investigate the complex-valued Koopman eigenvalues λ_p. Notably, these are the only quantities in Eq. 4.33 that change with time with $t/\Delta t$ in the exponent. The left graph in Fig. 4.27 shows the point distribution of all $P = 500$ Koopman eigenvalues on the complex plane. Importantly, most of the eigenvalues locate around the unit circle, where the magnitude of the eigenvalue $\lambda^{t/\Delta t}$ remains constant in the limit of $t \to \infty$. Since the system is linear, the stability criteria apply (cf. Table 2.1 on page 15). Eigenvalues inside the circle decay towards zero over time and, therefore, account for transient components — the additive term $\mathbf{v}_p \lambda_p^{t/\Delta t} \xi_p(\mathbf{z})$ in Eq. 4.33 decreases to zero.

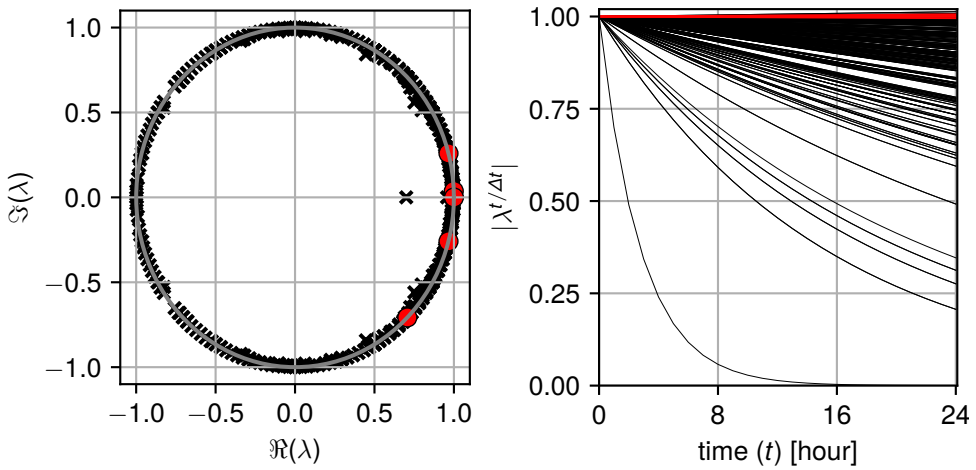


Figure 4.27: Left: The distribution of the Koopman eigenvalues $\lambda_p \in \mathbb{C}$ on the complex plane with the unit circle; $|\lambda| = 1$. Right: The magnitude of the eigenvalues over the prediction horizon. The red highlighted eigenvalues and curves correspond to Koopman eigenfunctions that are included in Fig. 4.28. Adapted from Lehmberg et al. [2021].

The right graph in Fig. 4.27 shows the eigenvalues' magnitude over the 24 hour prediction horizon. From the graph I infer the overall stability of the system in the specified prediction horizon of 24 hours. The plots highlight how many terms remain relevant over the prediction horizon. Since most eigenvalues have a magnitude of approximately one in the Melbourne EDMD model, the contribution of most eigenvalues remains significant. This highlights that most of the components attribute to periodic patterns in the data, which is expected in the (idealized) periodic traffic patterns.

Some eigenvalues also have a magnitude of greater than one and therefore classify as unstable for $t \to \infty$. In the right graph, some curves (including the red ones) increase slightly. However, because the largest computed eigenvalue is $|\lambda_{\text{max}}| \approx 1.0005$, this does not become "critical" in the considered time horizon. The exponential increase does not yet lead to rapid absolute increases in the values. It is also possible to project these *slightly* unstable eigenvalues onto the unit circle to obtain a model that is guaranteed to be stable. However, according to the right graph of Fig. 4.27 this is not necessary for the 24 hour prediction setting.

In terms of the model's stability, the EDMD framework largely depends on the dictionary choice, which for the Melbourne data is the composed data transformation $\mathbf{g}(\mathbf{x})$ in Eq. 4.24. Acknowledging the additional challenges for system identification on the real-world data (including noise, unobserved events and concept drifts) it is evident that the selected dictionary is robust to these factors and can extract and describe essential periodic patterns. As highlighted in the geometric analysis, the function basis $\{\psi_p(\mathbf{y})\}_{p=1}^{P}$ is ordered in terms of frequency on the manifold. This allowed truncating high-frequency and likely noise corrupted terms in the dictionary of the EDMD model.

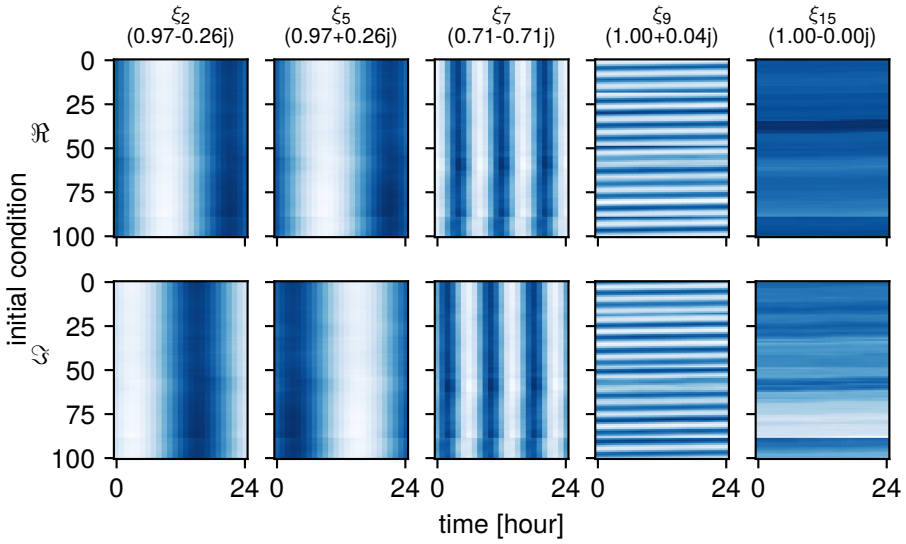


Figure 4.28: The visual selection of five (approximate) Koopman eigenfunctions $\xi_p(\mathbf{z})$ in the 24 hour prediction setting. The associated eigenvalue is stated in brackets. The displayed data contain the first 100 initial conditions of the test set. The top and bottom rows display the real and imaginary parts of the eigenfunctions respectively. The color ranges are scaled between the minimum and maximum values found in either the real or imaginary part of each function. Adapted from Lehmberg et al. [2021].

Finally, I visualize the complex-valued Koopman eigenfunctions ξ_p, that can reveal intrinsic system properties [Mauroy and Mezić, 2016]. In the linear dynamical system representation these provide the state in which the system is evolved (cf. Eq. 4.25 – 4.26).

Fig. 4.28 includes a selection of Koopman eigenfunctions $\xi_p(\mathbf{z})$, based on visually interesting features. The argument $\mathbf{z}$ is obtained from a time series of original sensor measurements with 169 samples (to perform the time delay embedding). In the figure, the eigenfunctions are evaluated on a sliding window of size 169 over the test data and the values are then projected again on the 24 hour prediction horizon. Essentially, this corresponds to how the initial states change over time.

Unlike for the DMAP coordinates ψ_p in Fig. 4.24, there is no immediate order for the Koopman eigenfunctions. Most relevant eigenpairs for the prediction have an associated eigenvalue that is approximately one in magnitude $|\lambda_p| \approx 1$. One way to order the eigenfunctions is to normalize the modes to unit length $\|\mathbf{v}\|_p = 1$, and take the absolute value of the p-th component of the (vectorized) Koopman eigenfunctions $\left|\xi_p(\mathbf{z}_1^{(c)})\right|$ as a measure for the importance; see Manojlović et al. [2020] for a detailed description (in the EDMD model this option is available through the flag `sort_koopman_triplet=True`).

The first eigenfunction, ξ_1, which is not displayed in Fig. 4.28 has a constant real part and is zero in the imaginary part.

Notably, the subsequent eigenfunctions ξ_{2-5} (only 2 and 5 are displayed in Fig. 4.28) strongly resemble the first two DMAP coordinates in Fig. 4.24. The sinusoidal pattern and shift show up in both the real and imaginary part of the function, with repeated directions. Other patterns, such as higher oscillations (ξ_7), weekly (ξ_9) and seasonal (ξ_{15}) patterns also re-appear in the Koopman eigenfunctions. Overall, this highlights that the geometric state representation in the specified EDMD dictionary provides a suitable basis to approximate the Koopman operator. Moreover, it highlights that the Koopman eigenfunctions themselves align to the inferred system geometry [Mezić, 2020]. A plot containing the first 20 eigenfunctions is located in the supplementary material of Lehmberg et al. [2021][4].

For the Melbourne analysis, I showed that the established operator setting and *datafold* are promising candidates to analyze high-dimensional time series data. The constructed EDMD model is robust to common challenges of real-world data. Despite the linear structure of the model, it is still capable of collectively and accurately forecasting multiple sensors with diverse characteristic traffic patterns. Importantly, the model is accessible and explainable through the analysis of the intrinsic components that relate to the Laplace-Beltrami and Koopman operators.

It is not surprising, yet reassuring, that the dominant directions in the geometric analysis revealed daily and weekly patterns. However, there was no explicit notion of what a day or week is. Similar to the pendulum in Section 4.1, these coordinates are intrinsic to the system and are likely related to "physically meaningful" quantities of Melbourne pedestrian traffic. In this representation, it could be possible to compare the patterns of different traffic systems, for example using data from another city.

A useful property is that the model's prediction stability can be quantified and comes with the guarantee of a linear dynamical system. Interestingly, the Koopman operator captures the principal patterns of all 11 sensors on a single spatio-temporal basis of the eigenfunctions. I hypothesize that it would be possible to describe a new (complex-valued) function family that is dedicated to traffic systems similar to the Melbourne data, that mimics the extracted Koopman eigenfunctions. That would avoid extracting the eigenfunctions again and would instead build on the available patterns to describe a linear system in which only the Koopman modes reconstruct the concrete traffic patterns. In Section 5.2, I highlight other promising future directions.

4.4 Benchmark analysis of *rdist*

During my thesis research, I also followed a secondary goal: Accelerate kernel-based methods (in particular DMAP). The approach is to efficiently compute a sparse δ-range distance matrix, which decreases the computational requirements in memory and subsequent operations, such as computing eigenpairs.

I highlight the necessity of a distance matrix when constructing a neighborhood graph as an empirical representation of a geometry in data in Sections 2.4.1 and 2.4.2.

[4] see the repository at https://github.com/datafold-dev/paper_modeling_melburnians

I also describe the "project-search-pullback" idea in Algorithm 1 (page 87) and its implementation in ***rdist***.

The results in this section differ from the previous three sections: Here I perform a benchmark analysis of ***rdist*** against alternative state-of-the-art implementations to construct an exact sparse δ-range distance matrix. Unlike processing time series data, I consider the data to be static. The underlying idea of a sparse matrix to reduce computational requirements (apart from geometric reasons) is to avoid the computation and storage of distance values that lead to negligible kernel values,

$$\mathcal{K}(d(\mathbf{x}_i, \mathbf{x}_j)) \begin{cases} > 0 & \text{relevant (small distance)} \\ \approx 0 & \text{negligible (large distance)} \end{cases} \tag{4.34}$$

where $d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_j - \mathbf{x}_i\|_2$ is the Euclidean distance function between two points. I only consider the common case of exponentially decaying kernels $\mathcal{K}$, such as the standard Gaussian kernel $\exp(^{-d(\mathbf{x}_i,\mathbf{x}_j)}/_{2\varepsilon})$. In this setting the objective is to avoid storing and processing pairs with a *large* distance that exceeds the specified radius $d > \delta$.

Diffusion Maps is an integral part of the main setting of my thesis to obtain an intrinsic state representation in geometric meaningful coordinates. It is also the most computationally expensive component within the model architecture. When developing ***rdist*** the goal was to accelerate operator-based system identification and to scale better with larger datasets. However, with the ongoing development of ***datafold***, I also implemented routines to systematically perform cross-validation and parameter optimization in EDMD; see class `EDMDCV` in Section 3.5.2. These allow optimizing the kernel values against a validation error. However, these routines only became available after I had implemented ***rdist***.

An outcome of the parameter optimizations was that larger kernel bandwidth values led to lower validation errors — for the pendulum see Fig. 4.8, for bus station Fig. 4.14 and for the Melbourne sensor data Table 4.3. Unfortunately, such large kernel bandwidth values diminish or even contradict my objective of accelerating DMAP with a sparse kernel matrix. This is because fewer kernel values are negligible (i.e. near zero as per Eq. 4.34) and the cut-off radius, δ, needs to be increased accordingly. Therefore, I did not include ***rdist*** for the data scenarios. Besides the reduced applicability, it has the drawback of introducing an additional parameter and makes the overall model structure more complex.

Nevertheless, the computation of sparse δ-range distance matrices is a fundamental task in computer science and a core element of many machine learning approaches. The underlying "project-search-pullback" idea and benchmark results of ***rdist*** are therefore still interesting. In fact, DMAP is a versatile method that is used for many different tasks, such as clustering [Coifman and Lafon, 2006b] or data fusion [Dietrich et al., 2021b; Lafon et al., 2006]. Smaller bandwidth parameters may be more suited in these other tasks. This is also suggested by the fact that all software projects that provide an implementation of DMAP in Table 3.4 (page 96) support a sparse kernel matrix (δ-radius or k-NN).

4.4.1 Benchmark setting

The goal of the benchmark is to compare *rdist* against other state-of-the-art algorithms that compute a sparse δ-range distance matrix. The search has to be exact so that all distance values $d(\mathbf{x}_i, \mathbf{x}_j) \leq \delta$ are stored. The main assumption of *rdist* is that the given points $\mathbf{x} \in X$ are in an high-dimensional ambient space $\mathbb{R}^N$, but intrinsically sampled on an (unknown) manifold with dimension $n \ll N$.

As highlighted in Section 3.3.2.2, *Rdist* can accelerate *existing* δ-range nearest neighbor algorithms, by performing the additional steps "project" and "pullback". Throughout the benchmark analysis, I use the kd-tree implementation within *rdist* from the high performance C++ library *mlpack* for the "search" task, already highlighted in the Algorithm 1. This choice is a result of experimentation within the benchmarks of the next two sections. A key feature of *rdist* is that once a faster algorithm or implementation becomes available it is easy to exchange the search algorithm within *rdist*.

For the benchmark, I only consider computing a *pairwise* distance matrix. This operation usually takes place during the training phase of a model. It is therefore the critical operation for the model's construction time. The pairwise distance matrix scales quadratic in size. However, depending on the sparsity structure, induced by the radius δ, only a fraction of pairwise distance values are stored. Depending on the method, not all pairwise distances need to be computed to validate whether the distance is smaller or larger than the radius. In particular, tree-based data structures are widely adopted to perform neighborhood search tasks. This is because the data is stored in a way that already gives a coarse neighborhood relation. Within the structure, only distance of point pairs in nearby "leaves" in the tree structure need to computed [Muja and Lowe, 2014].

For the analysis, I include six state-of-the-art and widely-used search algorithms, summarized in the Table 4.5. Like *rdist*, all included algorithms perform an exact δ-range nearest neighbor search. Note that each implementation of the table could potentially also be used within *rdist*.

The source code to perform the benchmarks is available within the *rdist* repository in the Supplementary Material. The benchmark analysis in the next two sections is executed on a Linux server with four processors of Intel® Xeon(R) Gold 6230 CPU of 2.1 GHz (x86-64 architecture) and a total of 130 GiB of computer memory (RAM) available. I disabled all parallelization and performed the all computations on a single processor.

Table 4.5: Overview of tree-based algorithms and libraries included in the benchmark analysis. The last column refers to the programming language ("Python/C" indicates the high-level front-end and low-level implementation).

Software	Structure	Short description	Language
FLANN[2]	kd-tree	Fast library for approximate nearest neighbors (FLANN) (also provides an exact search)	C++
nanoflann[1]	kd-tree	fork of *FLANN* for exact searches, specialized for point dimensions in $\mathbb{R}^2$ or $\mathbb{R}^3$	C++
SciPy[3]	kd-tree brute force	standard implementation of Python scientific computing stack. all distance pairs are computed for reference	Python/C
mlpack[4]	kd-tree covertree balltree	tree based data structures processed with dual-tree algorithms [Curtin et al., 2013]	C++
scikit-learn[5]	balltree kd-tree	tree-based data structures within the Python machine learning stack	Python/C

[1]Blanco and Rai [2021]; [2]Muja and Lowe [2009]; [3]Virtanen et al. [2020]; [4]Curtin et al. [2018]; [5]Pedregosa et al. [2011]

4.4.2 Generated data: Swiss-roll

In this first benchmark, I make use of a generated dataset. While data in such a "clean" form are not very representative of real-world datasets, it gives a guarantee that the manifold assumption is fulfilled. The generated data allows both the number of samples and number of point dimensions to be varied to see how each algorithm in Table 4.5 scales.

I generate data that is sampled on the swiss-roll manifold, which I used as an example geometry throughout the Sections 2.4.1 and 2.4.2 (cf. Fig. 2.6, page 42). I use the function `make_swiss_roll(n_samples)` of *scikit-learn* [Pedregosa et al., 2011], which samples three-dimensional data that is distributed on the two-dimensional swiss-roll. For higher ambient point dimensions of the data $N > 3$, I project the data with a random and unitary matrix into the target dimension. The intrinsic manifold dimension of $n = 2$ is maintained.

The benchmarks for the two scaling cases are plotted in Fig. 4.29 for each algorithm of Table 4.5 (a csv-file in tabular form is available in the Supplementary Material). For both cases I set the cut-off radius to $\delta = 2.5$. The memory requirements increase when I scale the number of samples: Both the size of the matrix and the number of neigh-

bors per point increase, as more samples are generated within the neighborhood radius of each point. In contrast, when I scale the ambient point dimension N, the sparsity structure and size of the distance matrix remains constant throughout the increase. I parameterized *rdist* with a contraction bound of $\gamma = 0.9$, which leads to a projection into a three-dimensional space (variable R in Algorithm 1 on page 87).

In Fig. 4.29 (top graph), when scaling the number of samples, the brute force algorithm is already out-of-memory after the first increase. A dense matrix would require $(2 \cdot 10^5)^2 \cdot 8\text{byte} = 320\text{GB}$ (8byte for the storage of a double precision floating point). The considered algorithms for computing a sparse distance matrix are able to scale with the increasing number of samples. However, there are larger discrepancies between the computational runtime. The algorithm with the largest runtime is the kd-tree of *SciPy*. At $6 \cdot 10^5$ samples, the algorithm is terminated because it takes longer than the maximum given time of 2 hours.

The fastest algorithm is *rdist*, which performs substantially faster than *FLANN* (27× faster), *nanoflann* (23× faster) and the algorithms from *scikit-learn* (4× and 8× faster for the balltree and kd-tree respectively). All improvements refer to the case of 1.2 million samples and an ambient point dimension of $N = 500$.

The difference of *rdist* to the algorithms in *mlpack*, however, are less significant; it is 1.94× faster than the covertree and 1.09× faster than the balltree. The difference to the kd-tree is particularly interesting, because *rdist* uses the same implementation for the internal search. Despite the additional processing steps "project" and "pullback", *rdist* is 1.7× faster compared to using the kd-tree directly. The library *mlpack* uses a dual-tree approach, which presumably leads to superior computational performance in comparison to other tree-based implementations [Curtin et al., 2013].

When scaling the point dimension, the benchmark is similar; *rdist* is again fastest, closely followed by the tree-based algorithms of *mlpack*.

The balltree or covertree implementation of *mlpack* could be also suitable choices to be used in *rdist*. However, I decided for the kd-tree. A reason against the covertree is that the method also comes with a manifold assumption to accelerate the nearest neighbor search [Beygelzimer et al., 2006]. The computational benefit of covertree seems to be "removed" when using it within *rdist*. Another reason why I decided against using the other two alternatives in *rdist* is that both algorithms are less robust for the real-world datasets, which I highlight in the next section.

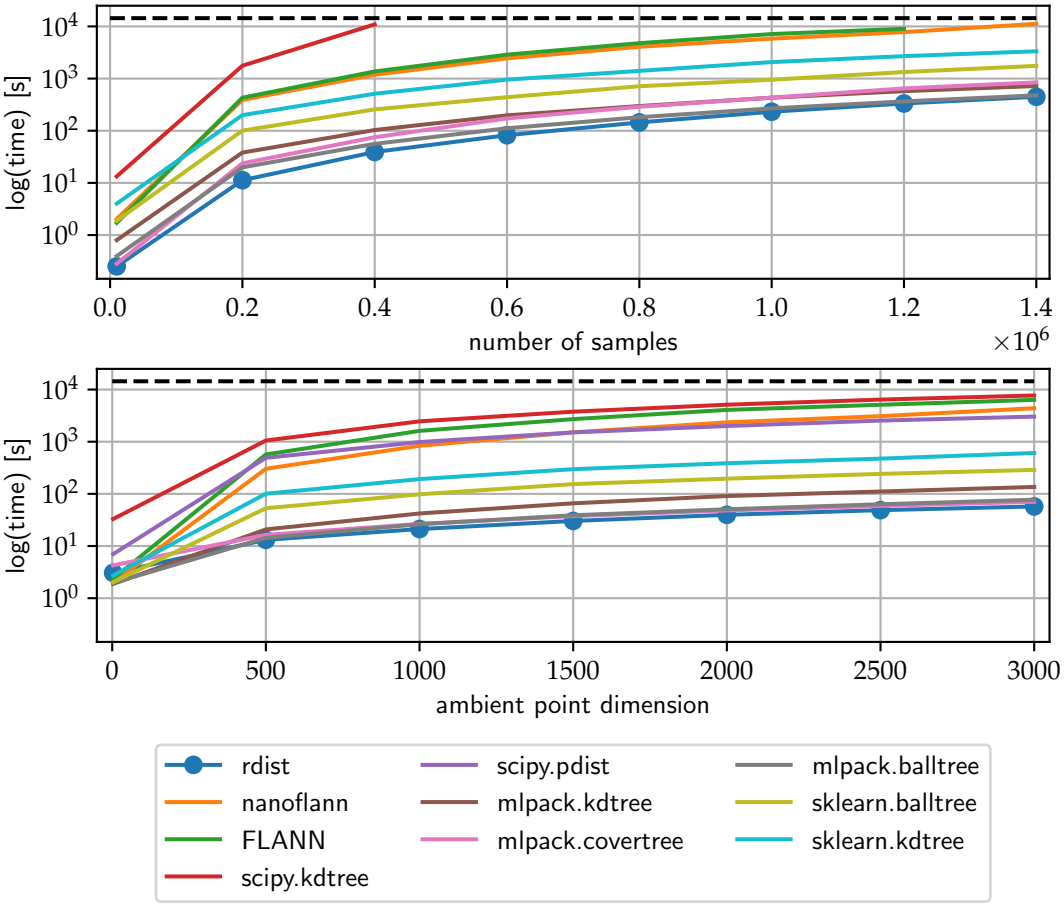


Figure 4.29: Results of the swiss-roll benchmark analysis for *rdist* and the nine other algorithms of Table 4.5. The top graph scales the number of samples with a constant ambient point dimension of $N = 500$ (the first sample is at 10000 samples). The bottom graph scales the ambient point dimension with a constant number of samples of 50000 (the first sample is at ambient point dimension $N = 3$). The dashed line at 7200 seconds is the time after which an algorithm is terminated.

4.4.3 Real-world datasets

I continue to benchmark *rdist* on a selection of real-world datasets. I only include the algorithms of *mlpack*, because these provide the fastest implementation of exact tree-based nearest neighbor searches. The other algorithms within Table 4.5 perform much worse and were mostly terminated after 2 hours.

Table 4.6 summaries the data selection of five real-world datasets. Each dataset has different characteristics in terms of the number of samples and the ambient point dimension. Because the data are real-world observations, the exact data characteristics are unknown and the manifold assumption is no longer guaranteed to be fulfilled. For each dataset, I specified a δ-radius, which determines the number of nonzero

(nnz) elements in the final distance matrix. The sparsity degree can be obtained with $^{\text{nnz}}/_{\text{nr samples}^2}$.

Rdist comes with a parameter γ, which describes the contraction bound to determine the dimension of the projection. Instead of optimizing this parameter, I include three version of *rdist* with $\gamma \in \{0.7, 0.8, 0.9\}$.

Table 4.6 shows that *rdist* is faster than the *mlpack* algorithms for four of the five datasets. For the popular MNIST dataset the acceleration factor to the kd-tree is 7.66× and to the covertree 12.46×. The only case in which *rdist* falls behind is the CovType dataset with a dramatic deceleration factor of about 0.16. Since the covertree also performs much worse than the balltree and kd-tree, presumably the manifold assumption does not hold for this dataset.

Another important factor in the comparison is to look at the robustness of the algorithms. While for *rdist* only one dataset is terminated due to out-of-memory (YearPredictionMSD with $\gamma = 0.7$), the *mlpack's* balltree fails for the first three datasets and the covertree fails for the last. Notably, for the generated swiss-roll dataset, the balltree is closest to *rdist* in terms of runtime, but now in the real-world data setting is less robust. Only the kd-tree and the two configurations of $\gamma = \{0.9, 0.8\}$ in *rdist* successfully compute a distance matrix for all datasets.

Since *rdist* includes the kd-tree of *mlpack*, it is interesting to measure the factor of how much the additional processing steps "project" and "pullback" accelerate the original kd-tree implementation. This is highlighted in Fig. 4.30. The largest acceleration is achieved for the FashionMNIST dataset, where *rdist* outperforms the original kd-tree implementation by a factor of about 10×.

Overall, I could show that the algorithmic idea of the "project-search-pullback" is a promising approach to accelerate δ-range distance matrix computations. Based on the implementation in *rdist*, I expect algorithmic improvements that could even further reduce the memory footprint and accelerate the computation. For example, in the current implementation the "search" and "pullback" are fully separated in *rdist* because of the communication between *rdist* and *mlpack*. However, a *combined* search and pullback per point pair can further decrease the memory footprint to essentially the final memory requirement of the distance matrix. While *rdist* is not a central result in my thesis, it is still a promising aspect that I suggest to follow up in future work.

As for the acceleration of DMAP within the main operator-informed system identification setting, new ideas are required to better scale with the length of time series (i.e. the number of points in the dataset). For example, Shen and Wu [2020] describe "RObust and Scalable Embedding via LANDmark Diffusion" (ROSELAND) as a promising variation of DMAP, which avoids computing a full pairwise kernel matrix.

Table 4.6: Benchmark results of *rdist* (with three different contraction bounds γ) against tree-based algorithms from *mlpack* for five datasets. The values for each algorithm are in seconds and the fastest algorithm of each dataset is highlighted in bold. "n/a" signals that the algorithm did not succeed, either because it is out-of-memory or required more than 2 hours. (nnz = number of non-zero values in the final distance matrix in millions [M])

Nr	Dataset	Characteristics of dataset				runtime *rdist*			runtime *mlpack*		
		samples	N	δ-range	nnz [M]	$\gamma = 0.9$	$\gamma = 0.8$	$\gamma = 0.7$	kd-tree	ball-tree	cover-tree
1	MNIST	70000	785	1600	47	525	**400**	468	3067	n/a	4987
2	FashionMNIST	70000	784	1370	49	**267**	297	424	2713	n/a	2936
3	PedTraj	109889	800	850	120	**256**	347	498	322	n/a	370
4	CovType	581012	54	160	69	241	218	217	**35**	40	118
5	YearPredictionMSD	515344	90	600	110	**804**	1168	n/a	4232	4414	n/a

[1]LeCun and Cortes [2010]; [2]Xiao et al. [2017]; [3]Majecka [2009], [4]Blackard et al. [1998]; [5]Bertin-Mahieux et al. [2010]

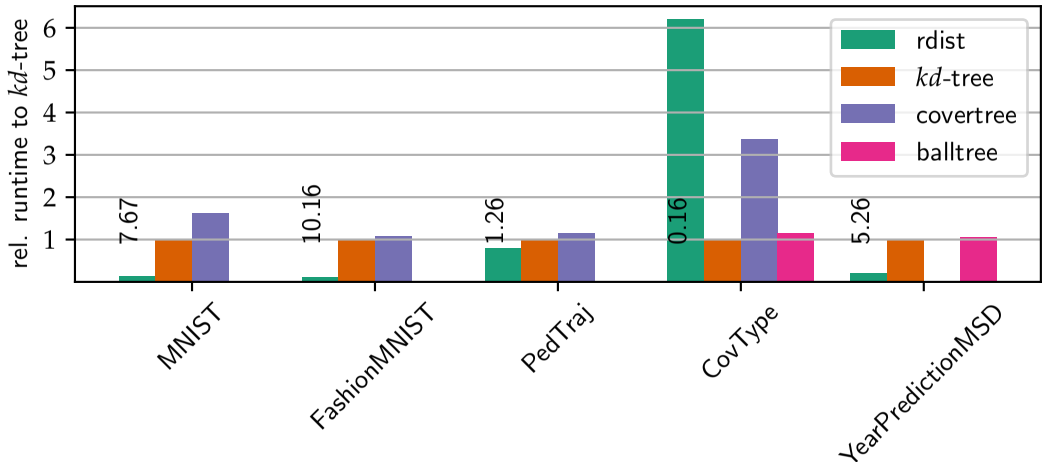


Figure 4.30: Comparison of the benchmark results in Table 4.6 for all five datasets. The bars show the relative runtime of each algorithm compared to kd-tree of *mlpack* and the numbers indicate the acceleration or deceleration factor of *rdist* compared to kd-tree.

4.5 Summary

In this chapter, I performed three system identifications and a benchmark analysis on simulated and real-world data. This represents the second part of my thesis contribution. For each case, I used the central methodologies in the scientific context of Chapter 2 that were transferred to a machine learning perspective in Chapter 3.

The three system identification scenarios all differ in terms of their data origin, dynamics and complexity. I performed the analysis in an equation-free workflow, based solely on time series collection data. The operators stored in a model describe intrinsic geometric and dynamic coordinates. This gives insight into the model's characteristics and also the hidden system, which is beyond a common error analysis. I ordered the three systems according to their complexity:

1. For the pendulum system I generated time series from a governing first order ordinary differential equation. I intentionally chose the (intuitive) Cartesian coordinates which are only a partial observation of the dynamical system. With the composed transformation of time delay embedding and Diffusion Maps I extract a new functional state representation that both reconstructs the dynamics and forms a geometrically aligned basis. Within these states, I selected suitable variables that are qualitatively equivalent to the angular coordinates as the true and hidden system variables. Moreover, the function basis was well-suited to approximate a Koopman operator in a linear dynamical system. This allowed me to accurately predict the entire (nonlinear) state evolution of the pendulum from an initial condition until it reaches the steady state.

2. In the bus station system I simulated and averaged stochastic time series from a microscopic pedestrian simulator. The governing system equations are no longer available. The generated time series described three crowd density values, which are the result of a scale transition from the microscopic simulator. In the analysis, I could show that the surrogate model accurately reconstructs and interpolates the nonlinear and transient crowd density evolution, parametrized with the number of agents in the bus. Moreover, I showcased the strong computational advantage compared to the original simulator. The operator-informed model acts as a data-driven surrogate, which circumvents the microscopic state evolution. In a forward uncertainty quantification, the constructed surrogate model only requires a fraction of the computational resources. The advantage is particularly relevant for environments in which the probability distributions of uncertain parameters are changing.

3. For the Melbourne pedestrian analysis, I used real-world time series from 11 sensors in the City of Melbourne, Australia. Each sensor has distinct traffic patterns, which collectively describe the pedestrian traffic on a city-scale. I could show that the operator-informed model can accurately predict all sensors collectively and robustly deal with noisy and interrupted measurements. I performed all analyses in a 24 hour prediction setting on a large chronologically separated test set. With the approximated operators contained in the model, I gained insight into the system

and model. I identified leading geometric coordinates which order and separate the system's time scales into daily, weekly and seasonal patterns. Altogether the state describes a common state space on which all 11 traffic patterns are captured. The quality to linearize the system dynamics made it suitable to approximate the Koopman operator in a matrix. All distinctive traffic patterns were encoded in a single dynamical system with a matrix of 500 rows and columns. The spectral components of the linear system expose easy-to-interpret model characteristics, such as the stability over an arbitrary prediction horizon (including the limit of $t \to \infty$).

In each scenario I could show that my proposed consolidation of the numerical frameworks time delay embedding, Diffusion Maps and Extended Dynamic Mode Decomposition could accurately identify the dynamical system within an interpretable model. *Datafold* was essential to the data analysis and provided the necessary functionality of data-driven modeling. There is currently no other software available that has the capabilities to perform this analysis. For each scenario, I could optimize the parameters involved by minimizing the validation error and generalize the model to new prediction settings (`EDMDCV` in Section 3.5.2). Such procedures are well-established in machine learning but are often unaddressed in operator-based modeling. Each of the three data applications required processing a collection of time series — made possible with the data structure `TSCDataFrame` in Section 3.3.1.1 — either because of the sampling procedure or because of intervals of missing data.

In addition to the three system identification analyses, I included a benchmark analysis from *rdist* with both simulated and real-world data. I could show that the algorithmic idea "project-search-pullback" can accelerate existing state-of-the-art implementations for the computation of a sparse distance matrix. The task is a core element of many machine learning algorithms, particularly Diffusion Maps (DMAP) as a kernel-based method.

5 Conclusion and future directions

In this last chapter, I reflect on the methods and results of my thesis. Section 5.1 summarizes my contribution to research for data-driven modeling of dynamical systems. I highlight the value that my software and analysis provide to the scientific community. Finally, in Section 5.2, I highlight promising directions for future research. For reference, I repeat my two overarching research questions:

Research questions

1. How can the operator-informed modeling approach for system identification — comprising the Laplace-Beltrami operator and Koopman operator — be translated into a machine learning perspective?
2. Can the setting from (1) provide insight into the model and (hidden) dynamical system by making use of the operators that store geometric and dynamical information?

5.1 Summary and conclusions

The main goal of my thesis was to examine and advance an operator-informed modeling approach to estimate a dynamical system from time series data and gain insight into the identified systems. In a concrete operator-based model I included three components having a different purpose: (1) time delay embedding to reconstruct partial measurement data, (2) the Laplace-Beltrami operator to obtain a geometrically informed function basis and (3) the Koopman operator for nonlinear system identification. Both operators are linear and store essential information about the geometry and dynamics of the system. I transferred this setting to a machine learning perspective with the two main numerical frameworks to approximate the operators: Diffusion Maps (DMAP) and Extended Dynamic Mode Decomposition (EDMD). I highlight that this numerical architecture has not been reported elsewhere in the literature (except by my paper Lehmberg et al. [2021]). However, there are deep connections established between the three components that justify this setting [e.g. Berry et al., 2013; Das and Giannakis, 2019; Giannakis, 2019]. The final model performs interpolations and predictions based on the system-intrinsic coordinates of the operators. In the vocabulary of operator theory the model is interpretable.

Scientific software for operator-informed system identification
With my **first research question** I focus on scientific software as an integral part of a data-driven modeling workflow. For operator-based methods there is rich literature available that has incrementally improved core numerical frameworks. However, scientific software that unifies these innovations is lacking. Without such software, the main model architecture within my thesis that consolidates the methods from different research lines was not possible. I transferred and organized the methods in *datafold*, which I conceptualized and implemented to answer the first research question. An essential part of this transfer was to consider aspects from traditional computer science and machine learning, such as interface design, class organization, computational efficiency, data structures or systematic parameter optimization on validation error. Ultimately, the software allowed me to address the concrete data applications covered within the second research question. At multiple steps, however, I could show that *datafold* overcomes limitations which I found in other (partial) software solutions (cf. Table 3.2, Table 3.4 and Table 3.7).

The main requirement of *datafold* is to provide high flexibility in the modeling workflow by being able to revise a model within the system identification loop (cf. Fig. 2.9). This flexibility also promotes future development because it is easier to exchange single components. *Datafold's* software architecture comprises three hierarchical layers that give orientation in the modeling workflow, described by the system identification loop. In the lowest layer, I contribute a data format to store time series collection data, for which no standardized and compatible solutions are available in the Python ecosystem. The data format was essential for generalizing the possible sampling schemes within the operator regression frameworks. All of the concrete data applications in my thesis comprised multiple time series and therefore relied on the data structure. On the second software layer, I specified methods that either extract temporal and/or spatial features from time series data. I exemplified these components to find intrinsic state representations with the methods involved in my main model approach — time delay embedding (temporal) and Diffusion Maps (spatial). The main method on *datafold's* third layer is the EDMD, which performs a nonlinear system identification by approximating the Koopman operator. I used a pipeline structure as a "meta-model" that allows flexible specification of the dictionary of one-to-many data transformations (acting as a basis to linearize the dynamics) and a specification of the method to perform the actual mode decomposition.

Within each layer, the data-driven methods are clearly separated and serve a dedicated purpose that is highlighted by mixin classes as a design pattern. Most methods in *datafold* integrate into Python's scientific computing stack, which allowed me to re-use existing and well-tested functionality that is managed by a large community. An important element here is the *scikit-learn* package because it provides core machine learning procedures. Overall, I relied on the flexibility in *datafold* for the model exploration, which guided me to the main operator-based architecture and allowed me to optimize the parameters for the concrete data applications.

Datafold provides a valuable gateway for researchers, practitioners and students who want to explore and apply operator-based methods for their own data analysis. The high degree of modularity allows each software component to be used on its own or as building blocks within larger settings that are covered by the numerical frameworks. The software fulfills high-quality standards as it is open-source, well-documented, and tested. I made use of *datafold* in my peer-reviewed publications [Lehmberg et al., 2020a, 2021], but the software has also been used by external research groups: within an uncertainty quantification setting [Upadhyay et al., 2021], for learning stochastic differential equations from microscopic simulations [Dietrich et al., 2021a] and for time series prediction [Papaioannou et al., 2021].

Another target group are researchers who want to perform algorithmic experimentation within *datafold*. The software's design makes it possible to set up and vary a complex data processing pipeline and analyze, validate and compare the effects of methodological and algorithmic changes. This was quality was used by a student group who investigated efficient sparse eigensolvers on a cluster Grad and Raith [2021]. Within my thesis, I could demonstrate and make use of the algorithmic experimentation by using a sparse distance matrix. While my goal was to accelerate the computation of Diffusion Maps — as the computationally most expensive part of the main operator setting — the model validation in the concrete data applications revealed unsuitable bandwidth settings in which sparse kernel matrices are no longer advantageous. Despite this being a limitation within the concrete data analysis, the "project-search-pullback" idea as implemented in *rdist* offers a positive algorithmic contribution. This is because the computation of a sparse distance matrix is a fundamental operation that often appears in computer science and machine learning. I showed that *rdist* can accelerate an existing kd-tree implementation from the C++ library *mlpack* such that it outperforms the original algorithm and other state-of-the-art implementations for most real-world data settings (Table 4.6). This makes it a promising algorithmic result to follow up on and integrate the algorithm into other tasks.

Data analysis with operator-informed approach

For the **second research question** I explored the operator-theoretic approaches to perform "scientific machine learning" on concrete data applications with the aid of *datafold*. The spectral components of the two linear operators allow the user to gain insight into the model. These capabilities to interpret the model are beyond typical error evaluation. In my thesis, I covered three dynamical systems with distinctive temporal patterns and increasing complexity.

I started with an ordinary differential equation that describes a pendulum. By going through the main components of the operator-based model architecture I demonstrated with an "academic system" that I meet my goals. I first extracted geometric coordinates from time series in Cartesian states, that are qualitatively equivalent to the hidden, physically meaningful and well-defined angular pendulum coordinates. Secondly, I used these geometrically aligned coordinates to construct a Koopman operator-based model and demonstrated that it accurately predicts an entire test trajectory. The pen-

dulum example allowed me to highlight the workflow and functionality of *datafold* in a simple system setting. I would therefore recommend using this as an example when getting started with *datafold*.

For the second data application, the bus station scenario, I increased the system complexity to stochastic data. The time series are simulated from the pedestrian traffic simulator *Vadere*. Such simulations can be used as a tool to assess crowd safety by analyzing what-if scenarios. The main goal was to build an operator-informed and surrogate model that describes the dynamics of three (averaged) pedestrian density values as *macroscopic* system quantities. While the access to the data-generating system allowed me to freely "design" the state quantities, the governing equations were no longer available in this setting. This is because *Vadere* only describes the *microscopic* state evolution in which the simulated pedestrians interact. For the model construction, I systematically optimized the surrogate model's parameters by following the good practice of minimizing a cross-validated error. There is no clear evidence that cross-validation procedures, that are well-established in the machine learning community, have been adopted within the operator-based literature. In my opinion, the missing information on how the parameters were selected can jeopardize the results because it increases the risk of overfitting the available data. Ultimately, with the final model I was able to generate time series that interpolate the entire state space, from the initial state line to the attractor where all agents have left the simulation scenario (cf. Fig. 4.13 and Fig. 4.16).

I also demonstrated the computational superiority of the surrogate model: The linear structure and the scale transition of the surrogate model only require a tiny fraction of the runtime of *Vadere*. By performing an uncertainty quantification, I showed that the computational advantage remains even when the uncertain parameter distribution changes (the analysis is similar to Dietrich et al. [2018]). This is a clear advantage compared to other established approaches such as Polynomial Chaos Expansion. Because the underlying pedestrian simulator can be actively queried for new data if necessary, there is still scope to further reduce the computationally costly microscopic simulation when constructing a surrogate model.

As the last system identification analysis, I transferred the operator-informed model to an application with real-world data. In contrast to the previous two systems, there was no longer access to the data-generating system to manipulate data characteristics, such as the sampling rate or selection of state variables. The measurement states are noise-corrupted, have intervals of missing data and include unobserved factors. Furthermore, the state evolution is no longer transient but has near periodic patterns (with intermittent unobserved forcing) and is assumed to be ergodic. The dataset comprised 11 different traffic sensors in the City of Melbourne that count the pedestrians that pass a sensor. Each sensor has distinct traffic patterns and pedestrian volume.

To reconstruct essential dynamic information for intraday and weekly traffic patterns, it was necessary to embed the single measurements within a time window of the entire past week. This resulted in high-dimensional states which were then mapped onto the geometrically aligned eigenfunctions of the Laplace-Beltrami operator. These

define a common basis for *all* traffic sensors. Ultimately, this allowed me to analyze the data-generating system of Melbourne's pedestrian traffic and, similar to the pendulum case, analyze the inferred geometry that is presumably one-to-one to the hidden data-generating system. I visualized the principal geometric coordinates that cover the intraday, weekly and also seasonal patterns within the time series data. Importantly, the function basis also (approximately) linearizes the dynamics for all sensors within the Koopman operator perspective. The spectral components of the Koopman operator describe a standard time-invariant linear dynamical system that gives valuable insight into the identified system. Particularly relevant are the eigenvalues that are easy to interpret and describe the model's stability over the 24 hour (or any other) prediction horizon. Furthermore, I could visually show that the complex-valued Koopman eigenfunctions form similar patterns to the real-valued geometric DMAP coordinates (as the EDMD dictionary). This highlights the connection between the Laplace-Beltrami and Koopman operators.

Overall, the operator-based model has only a few "critical" parameters that influence the model's quality: the number of time delays, the kernel bandwidth and the number of DMAP coordinates to include in the EDMD dictionary. Each of these parameters can also regularize the model. For the Melbourne dataset, I could derive heuristics for each parameter by using *datafold's* cross-validation capabilities. I suggest that this parametrization is transferable to similar data applications that are dominated by intraday and weekly patterns. This also includes vehicular traffic, which is likely to show similar traffic patterns to the ones measured in the pedestrian sensors (i.e. a dominating working-week pattern).

With the Melbourne sensor data, I contributed an innovative method to advance traffic research. The results are also published in the highly ranked journal "Transportation Research Part C: Emerging Methodologies" (impact factor of 8.09 at the time of publication) [Lehmberg et al., 2021]. This reinforces the potential of the operator-based approach (and *datafold*) as a promising method to analyze traffic flows in a smart city setting. Accurate 24 hour predictions can help to describe "usual traffic" and identify traffic abnormalities to allocate security resources effectively. The operator-based approach complements the commonly applied methods: neural networks and statistics-based methods. While the model selection between these two model types often represents a trade-off between accuracy and interpretability, the operator-informed approach has the potential to combine the qualities of both approaches and produce accurate and interpretable models that are capable of describing high-dimensional systems. However, in future work, it is important to investigate and compare accuracy measures in fair benchmark settings.

Final conclusion

In my thesis I explored an operator-informed approach to identify dynamical systems from multivariate time series collection data. The emerging methodology complements other approaches from statistics or "traditional" machine learning. *Datafold* fills an important gap by transferring numerical frameworks to approximate the Laplace-

Beltrami and Koopman operators as central components in my model architecture. While these frameworks are often used in literature, previously available software was inadequate for integrating them into a single data-processing pipeline. Furthermore, *datafold* includes established machine learning procedures with which a model can be revised and optimized within the system identification loop. *Datafold's* design lays the groundwork for ongoing transfer and unification of algorithmic improvements for operator approximation. In the data scenarios, I focused on traffic and mobility as a discipline that is often confronted with systems for which no governing equations are available. Moreover, observational data often only includes partial system information and is noise-corrupted. As a novel contribution to pedestrian traffic research, I transferred the operator-informed approach to such challenging settings. I showed that the models can accurately forecast different spatio-temporal traffic patterns. I also highlighted the usefulness of the approach by extracting system-intrinsic and interpretable coordinates from which a model compiles its traffic predictions. Ultimately, my thesis provides an approach and software to increase scientific understanding in data-driven modeling, including for scenarios where complex and inaccessible models are often adopted.

5.2 Future work

Both my contribution of *datafold* and the two traffic analyses provide much scope for future research. To keep the list concise I only present directions that I find most promising. For each direction, I sketch the realization in *datafold* and a possible exploration for data-driven applications with a focus on traffic systems.

- **Integrate system input and output quantities**

  For the Melbourne sensor data, I pointed out that there are unobserved effects that influence the traffic state, such as weather, public holidays or accidents. The EDMD can be adapted to account for these external forcing terms with an additional input [Mauroy et al., 2020; Proctor et al., 2018; Williams et al., 2016]. For the traffic system, this would allow the Koopman matrix to be "cleaned" by separating the external forcing effects into an additional system matrix.

  In *datafold* the interface would need to be extended to an additional time series input next to the standard measurement time series (commonly denoted with `X`). A prediction with additional input can only be performed as long as the input values are available over the prediction horizon. While some quantities are easy to obtain (e.g. public holidays), other quantities require additional forecasts, such as the weather from the official forecast. Other events that act on the system are nearly impossible to predict (e.g. accidents) and information can only be included in hindsight to describe historic data.

  The additional system input also leads to the vast topic of system control. This is particularly interesting for many engineering systems. In fact, the topic receives a

lot of attention in the Koopman operator research, because the linear model structure makes it advantageous to use the well-founded linear control theory. For an exhaustive book on the topic I refer to Mauroy et al. [2020].

It is also possible to attach additional information to the model output. For example, for the Melbourne sensor data, it is likely that the discovered intrinsic patterns are also suitable for *additional* sensors not included in the data selection. By manipulating the Koopman modes in the linear system formulation it is easy to capture new sensors that make use of the common basis of the sensors within the training set. This feature is interesting if a sensor has fewer data available, for example, because the sensor was only temporally installed. Computing a new Koopman mode only requires solving a linear system. Ultimately, this opens doors to further detail the overall city traffic, based on the measurements of the smaller set of selected sensors.

Another type of information that can be attached to the model's output are quantities that enrich a prediction. This can include uncertainty information of a prediction, such as the expected error based on the model error analysis of past data. For example, for the Melbourne data, it is much harder to predict the pedestrian traffic during the day than at nighttime. A similar approach to attaching such information is followed in the context of the kernel analog framework, which also connects to the Koopman operator theory. Alexander and Giannakis [2020, p. 2] suggest that "*with appropriate choices of the response observable, forecasts can be obtained not just for the conditional mean of an observed quantity, but also that quantity's conditional variance and higher-order moments, which are important for uncertainty quantification.*" These "attached observables" can be transferred to the EDMD framework in a straightforward manner.

- **Streaming setting**

  It is an inherent property of dynamical systems that data becomes available over time. Throughout my thesis, I processed the data in a batch fashion, in which all data is readily available. In contrast, in a streaming setting, a model is frequently updated over time once new data becomes available. A straightforward approach to make use of the current algorithms are "sliding windows", where the model is retrained by including new data and dropping old data in the window. However, there are also more sophisticated streaming settings that are suitable if the model training has to be efficient, such as when the data streams in with a high frequency. In such cases, it is necessary to avoid re-iterating pas data to reduce computational costs and efficiently update the model.

  There are DMD variants that adapt to such streaming settings [e.g. Hemati et al., 2014]. By transferring these to *datafold* the streaming-DMD can also be directly integrated in the EDMD framework (cf. Fig. 3.11). However, a streaming setting would also be required by all data transformations within the EDMD dictionary. For the main setting in my thesis, the time delay embedding is easy to adjust, but larger modifications are required for DMAP. Long and Ferguson [2019] propose a streaming setting with a landmark approach.

For traffic analysis, a streaming setting is more suitable for long-term and ongoing observations of a system where concept drifts are expected. In the Melbourne sensor data, for instance, a streaming setting could be adopted to detect and adapt to temporary changes in single sensors (e.g. due to construction works) or overall traffic. For example, during the SarS-CoV-2 pandemic the City of Melbourne mandated several lockdowns to reduce the spread of the virus. These concept drifts in the traffic could be used to explore the streaming capabilities.

- **Learning the EDMD dictionary**

  Throughout the thesis, I highlighted that a suitable dictionary choice within the EDMD framework is considered to be an open problem. This is because the choice depends on system characteristics that are often not well understood *a priori*. While I followed a generic function basis that connects to the data geometry within my thesis, there are also studies that aim to remove the burden of having to explicitly choose a dictionary. Li et al. [2017] use a neural network as a generic regressor within the EDMD framework to make the dictionary itself *trainable*. A drawback, however, is that this increases the overall model complexity and the state representation to linearize the dynamics is hidden in a neural network. Nevertheless, the final model is linear in the Koopman operator system and provides access to the spatio-temporal patterns in the spectral components.

- **Dataset benchmarks to consolidate methodologies**

  For the future development of *datafold* I recommend setting up a benchmark that includes multiple datasets that cover a variety of system characteristics. The three data scenarios that I analyzed within my thesis can serve as a starting point.

  A benchmark allows methodological results and algorithms to be quantified on these datasets. It therefore helps to consolidate the many incremental methodological improvements that are available in the literature — as exemplified by the numerous DMD and EDMD variants. I see this as a worthwhile step to structure and justify the methodological unification within *datafold* but also the operator-based modeling approaches.

  For the ongoing development of *datafold* the main objective would be to verify that any new methodological features are actually profitable by testing them on the benchmark. This could be either in terms of a higher accuracy or faster evaluation. Larger changes in the productive code of *datafold* should only be integrated after this benchmark phase.

My hope is that the development of *datafold* continues and enables new research — either by on the methodology side or by analyzing new data scenarios. In my thesis, I demonstrated the benefits of the operator setting within an equation-free system identification. However, there are many degrees of freedom that allow new data applications to be systematically explored to advance the exciting field of operator-based modeling.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016) TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467v2 [cs]*, page 19. https://arxiv.org/abs/1603.04467.

Alexander, R. and Giannakis, D. (2020) Operator-theoretic framework for forecasting nonlinear time series with kernel analog techniques. *Physica D: Nonlinear Phenomena*, 409:132520. doi:10.1016/j.physd.2020.132520.

Anaya, M. (2021) *Clean Code in Python - Second Edition*. Packt Publishing. ISBN 978-1-80056-021-5. https://learning.oreilly.com/library/view/clean-code-in/9781800560215/.

Andoni, A. and Indyk, P. (2006) Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science*, pages 459–468. IEEE. doi:10.1109/FOCS.2006.49.

Anzt, H., Bach, F., Druskat, S., Löffler, F., Loewe, A., Renard, B. Y., Seemann, G., Struck, A., Achhammer, E., Aggarwal, P., Appel, F., Bader, M., Brusch, L., Busse, C., Chourdakis, G., Dabrowski, P. W., Ebert, P., Flemisch, B., Friedl, S., Fritzsch, B., Funk, M. D., Gast, V., Goth, F., Grad, J.-N., Hermann, S., Hohmann, F., Janosch, S., Kutra, D., Linxweiler, J., Muth, T., Peters-Kottig, W., Rack, F., Raters, F. H. C., Rave, S., Reina, G., Reißig, M., Ropinski, T., Schaarschmidt, J., Seibold, H., Thiele, J. P., Uekerman, B., Unger, S., and Weeber, R. (2020) An Environment for Sustainable Research Software in Germany and Beyond: Current State, Open Challenges, and Call for Action. *F1000Research*, 9:295. doi:10.12688/f1000research.23224.2.

Arbabi, H. and Mezić, I. (2017) and Computation of Spectral Properties of the Koopman Operator. *SIAM Journal on Applied Dynamical Systems*, 16(4):2096–2126. doi:10.1137/17M1125236.

Askham, T. and Kutz, J. N. (2018) Variable Projection Methods for an Optimized Dynamic Mode Decomposition. *SIAM Journal on Applied Dynamical Systems*, 17(1):380–416. doi:10.1137/M1124176.

Åström, K. J. and Eykhoff, P. (1970) *System identification*. Technical Reports TFRT-7011. Department of Automatic Control, Lund Institute of Technology (LTH). https://portal.research.lu.se/en/publications/system-identification.

Avila, A. M. and Mezić, I. (2020) Data-driven analysis and forecasting of highway traffic dynamics. *Nature Communications*, 11(1):2090. doi:10.1038/s41467-020-15582-5.

Baer, M. (2021) findiff: Python package for numerical derivatives and partial differential equations in any number of dimensions. https://github.com/maroba/findiff. accessed: 1. Dec. 2021.

Bakker, C., Bhattacharya, A., Chatterjee, S., Perkins, C. J., and Oster, M. R. (2020) The Koopman Operator: Capabilities and Recent Advances. In *2020 Resilience Week (RWS)*, pages 34–40. doi:10.1109/RWS50334.2020.9241276.

Balasubramanian, M. (2002) The Isomap Algorithm and Topological Stability. *Science*, 295(5552):7a–7. doi:10.1126/science.295.5552.7a.

Beckers, T. and Hirche, S. (2020) Prediction with Gaussian Process Dynamical Models. *arXiv:2006.14551 [cs, eess]*. https://arxiv.org/abs/2006.14551.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011) Cython: The Best of Both Worlds. *Computing in Science & Engineering*, 13(2):31–39. doi:10.1109/MCSE.2010.118.

Belkin, M. and Niyogi, P. (2007) Convergence of Laplacian Eigenmaps. *Advances in Neural Information Processing Systems*, 19:8. https://dl.acm.org/doi/10.5555/2976456.2976473.

Belkin, M. and Niyogi, P. (2008) Towards a theoretical foundation for Laplacian-based manifold methods. *Journal of Computer and System Sciences*, 74(8):1289–1308. doi:10.1016/j.jcss.2007.08.006.

Belkin, M., Sun, J., and Wang, Y. (2009) Constructing Laplace Operator from Point Clouds in Rd. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1031–1040. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973068.112.

Bello-Rivas, J. M. (2017) jmbr/diffusion-maps. https://github.com/jmbr/diffusion-maps. accessed: 1. Dec. 2021.

Bengio, Y., Courville, A., and Vincent, P. (2013) Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828. doi:10.1109/TPAMI.2013.50.

Benosman, M., Mansour, H., and Huroyan, V. (2017) Koopman-operator Observer-based Estimation of Pedestrian Crowd Flows. *IFAC-PapersOnLine*, 50(1):14028–14033. doi:10.1016/j.ifacol.2017.08.2428.

Bergmeir, C. and Benítez, J. M. (2012) On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213. doi:10.1016/j.ins.2011.12.028.

Berry, T. and Giannakis, D. (2020) Spectral Exterior Calculus. *Communications on Pure and Applied Mathematics*, 73(4):689–770. doi:10.1002/cpa.21885.

Berry, T. and Harlim, J. (2016) Variable bandwidth diffusion kernels. *Applied and Computational Harmonic Analysis*, 40(1):68–96. doi:10.1016/j.acha.2015.01.001.

Berry, T. and Sauer, T. (2016) Local kernels and the geometric structure of data. *Applied and Computational Harmonic Analysis*, 40(3):439–469. doi:10.1016/j.acha.2015.03.002.

Berry, T. and Sauer, T. (2019) Consistent Manifold Representation for Topological Data Analysis. *arXiv:1606.02353 [math]*. https://arxiv.org/abs/1606.02353.

Berry, T., Cressman, J. R., Gregurić-Ferenček, Z., and Sauer, T. (2013) Time-Scale Separation from Diffusion-Mapped Delay Coordinates. *SIAM Journal on Applied Dynamical Systems*, 12(2):618–649. doi:10.1137/12088183X.

Berry, T., Giannakis, D., and Harlim, J. (2015) Nonparametric forecasting of low-dimensional dynamical systems. *Physical Review E*, 91(3). doi:10.1103/PhysRevE.91.032915.

Berry, T., Giannakis, D., and Harlim, J. (2020) Bridging Data Science and Dynamical Systems Theory. *Notices of the American Mathematical Society*, 67(09):1. doi:10.1090/noti2151.

Bertin-Mahieux, T., Ellis, D. P., Whitman, B., and Lamere, P. (2010) UCI Machine Learning Repository: YearPredictionMSD Data Set. https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD.

Beygelzimer, A., Kakade, S., and Langford, J. (2006) Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning - ICML '06*, pages 97–104, Pittsburgh, Pennsylvania. ACM Press. doi:10.1145/1143844.1143857.

Birkhoff, G. D. (1931) Proof of the Ergodic Theorem. *Proceedings of the National Academy of Sciences*, 17(12):656–660. doi:10.1073/pnas.17.2.656.

Bishop, C. M. (2006) *Pattern recognition and machine learning*. Information science and statistics. Springer, New York. ISBN 978-0-387-31073-2.

Blackard, J. A., Dean, D. J., and Anderson, C. W. (1998) UCI Machine Learning Repository: Covertype Data Set. https://archive.ics.uci.edu/ml/datasets/Covertype.

Blanco, J. L. and Rai, P. K. (2021) a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann. accessed: 1. Dec. 2021.

Bode, N. W., Chraibi, M., and Holl, S. (2019) The emergence of macroscopic interactions between intersecting pedestrian streams. *Transportation Research Part B: Methodological*, 119:197–210. doi:10.1016/j.trb.2018.12.002.

Bollt, E. M. (2021) Geometric considerations of a good dictionary for Koopman analysis of dynamical systems: Cardinality, "primary eigenfunction," and efficient representation. *Communications in Nonlinear Science and Numerical Simulation*, 100:105833. doi:10.1016/j.cnsns.2021.105833.

Bollt, E. M., Li, Q., Dietrich, F., and Kevrekidis, I. (2018) Dynamical Systems through Koopman Operator Eigenfunctions. *SIAM Journal on Applied Dynamical Systems*, 17 (2):1925–1960. doi:10.1137/17M116207X.

Bonabeau, E. (2002) Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(Supplement 3): 7280–7287. doi:10.1073/pnas.082080899.

Borchani, H., Varando, G., Bielza, C., and Larranaga, P. (2015) A survey on multi-output regression. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5(5): 216–233. doi:10.1002/widm.1157.

Boukerche, A. and Wang, J. (2020) Machine Learning-based traffic prediction models for Intelligent Transportation Systems. *Computer Networks*, 181:107530. doi:10.1016/j.comnet.2020.107530.

Bracha, G. and Cook, W. (1990) Mixin-based inheritance. *ACM Sigplan Notices*, 25: 303–311. doi:10.1145/97946.97982.

Broomhead, D. and King, G. P. (1986) Extracting qualitative dynamics from experimental data. *Physica D: Nonlinear Phenomena*, 20(2-3):217–236. doi:10.1016/0167-2789(86)90031-X.

Brunton, S. L. and Kutz, J. N. (2019) *Data-Driven Science and Engineering: Machine learning, dynamical systems, and control*. Cambridge University Press. ISBN 978-1-108-42209-3.

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

Brunton, S. L., Brunton, B. W., Proctor, J. L., Kaiser, E., and Kutz, J. N. (2017) Chaos as an intermittently forced linear system. *Nature Communications*, 8(1):19. doi:10.1038/s41467-017-00030-8.

Budišić, M., Mohr, R. M., and Mezić, I. (2012) Applied Koopmanism. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 22(4):047510. doi:10.1063/1.4772195.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., Vanderplas, J., Joly, A., Holt, B., and Varoquaux, G. (2013) API design for machine learning software: experiences from the scikit-learn project. *arXiv:1309.0238 [cs]*. https://arxiv.org/abs/1309.0238.

Bungartz, H.-J., Zimmer, S., Buchholz, M., and Pflüger, D. (2014) *Modeling and Simulation*. Springer Undergraduate Texts in Mathematics and Technology. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/978-3-642-39524-6.

Carter, E., Adam, P., Tsakis, D., Shaw, S., Watson, R., and Ryan, P. (2020) Enhancing pedestrian mobility in Smart Cities using Big Data. *Journal of Management Analytics*, 7(2):173–188. doi:10.1080/23270012.2020.1741039.

Castelvecchi, D. (2016) Can we open the black box of AI? *Nature News*, page 4. https://www.doi.org/10.1038/538020a.

Chan, F. and Pauwels, L. L. (2018) Some theoretical results on forecast combinations. *International Journal of Forecasting*, 34(1):64–74. doi:10.1016/j.ijforecast.2017.08.005.

Cheng, Z., Trepanier, M., and Sun, L. (2021) Real-time forecasting of metro origin-destination matrices with high-order weighted dynamic mode decomposition. *arXiv:2101.00466 [stat]*. https://arxiv.org/abs/2101.00466.

Chiavazzo, E., Gear, C., Dsilva, C., Rabin, N., and Kevrekidis, I. (2014) Reduced Models in Chemical Kinetics via Nonlinear Data-Mining. *Processes*, 2(1):112–140. doi:10.3390/pr2010112.

Chowell, G. (2017) Fitting dynamic models to epidemic outbreaks with quantified uncertainty: A primer for parameter uncertainty, identifiability, and forecasts. *Infectious Disease Modelling*, 2(3):379–398. doi:10.1016/j.idm.2017.08.001.

Clarkson, K. L. (2008) Tighter bounds for random projections of manifolds. In *Proceedings of the twenty-fourth annual symposium on Computational geometry - SCG '08*, page 39, College Park, MD, USA. ACM Press. doi:10.1145/1377676.1377685.

Coifman, R. R. and Lafon, S. (2006) Geometric harmonics: A novel tool for multiscale out-of-sample extension of empirical functions. *Applied and Computational Harmonic Analysis*, 21(1):31–52. doi:10.1016/j.acha.2005.07.005.

Coifman, R. R. and Lafon, S. (2006) Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5–30. doi:10.1016/j.acha.2006.04.006.

Coveney, P. V., Dougherty, E. R., and Highfield, R. R. (2016) Big data need big theory too. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2080):20160153. doi:10.1098/rsta.2016.0153.

Črnjarić-Žic, N., Maćešić, S., and Mezić, I. (2020) Koopman Operator Spectrum for Random Dynamical Systems. *Journal of Nonlinear Science*, 30(5):2007–2056. doi:10.1007/s00332-019-09582-z.

Curtin, R. R., March, W. B., Ram, P., Anderson, D. V., Gray, A. G., and Isbell, C. L. (2013) Tree-Independent Dual-Tree Algorithms. In *International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1435–1443. PMLR. https://proceedings.mlr.press/v28/curtin13.html.

Curtin, R. R., Edel, M., Lozhnikov, M., Mentekidis, Y., Ghaisas, S., and Zhang, S. (2018) flexible machine learning library. *Journal of Open Source Software*, 3(26):726. doi:10.21105/joss.00726.

Das, S. and Giannakis, D. (2019) Delay-Coordinate Maps and the Spectra of Koopman Operators. *Journal of Statistical Physics*, 175(6):1107–1145. doi:10.1007/s10955-019-02272-w.

Dawson, S. T. M. (2016) Characterizing and correcting for the effect of sensor noise in the dynamic mode decomposition. *Exp Fluids*, 57(3):19. doi:10.1007/s00348-016-2127-7.

de Silva, B. and Kaiser, E. (2021) PyKoopman. https://github.com/dynamicslab/pykoopman. accessed: 1. Dec. 2021.

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

Demo, N., Tezzele, M., and Rozza, G. (2018) PyDMD: Python Dynamic Mode Decomposition. *The Journal of Open Source Software*, 3(22):530. doi:10.21105/joss.00530.

Deyle, E. R. and Sugihara, G. (2011) Generalized Theorems for Nonlinear State Space Reconstruction. *PLoS ONE*, 6(3):e18295. doi:10.1371/journal.pone.0018295.

Dicle, C., Mansour, H., Tian, D., Benosman, M., and Vetro, A. (2016) Robust low rank dynamic mode decomposition for compressed domain crowd and traffic flow analysis. In *2016 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, Seattle, WA, USA. IEEE. doi:10.1109/ICME.2016.7552877.

Dietrich, F. (2017) *Data-Driven Surrogate Models for Dynamical Systems*. PhD Thesis, Technical University of Munich. https://mediatum.ub.tum.de/?id=1356533.

Dietrich, F., Köster, G., and Bungartz, H.-J. (2016) Numerical Model Construction with Closed Observables. *SIAM Journal on Applied Dynamical Systems*, 15(4):2078–2108. doi:10.1137/15M1043613.

Dietrich, F., Künzner, F., Neckel, T., Köster, G., and Bungartz, H.-J. (2018) Fast and flexible uncertainty quantification through a data-driven surrogate model. *International Journal for Uncertainty Quantification*, 8(2):175–192. doi:10.1615/Int.J.UncertaintyQuantification.2018021975.

Dietrich, F., Thiem, T. N., and Kevrekidis, I. G. (2020) On the Koopman Operator of Algorithms. *SIAM Journal on Applied Dynamical Systems*, 19(2):860–885. doi:10.1137/19M1277059.

Dietrich, F., Makeev, A., Kevrekidis, G., Evangelou, N., Bertalan, T., Reich, S., and Kevrekidis, I. G. (2021) Learning effective stochastic differential equations from microscopic simulations: combining stochastic numerics and deep learning. *arXiv:2106.09004 [physics]*. https://arxiv.org/abs/2106.09004.

Dietrich, F., Yair, O., Mulayoff, R., Talmon, R., and Kevrekidis, I. G. (2021) Spectral Discovery of Jointly Smooth Features for Multimodal Data. *arXiv:2004.04386 [cs, stat]*. https://arxiv.org/abs/2004.04386.

Ding, J., Tarokh, V., and Yang, Y. (2018) Model Selection Techniques: An Overview. *IEEE Signal Processing Magazine*, 35(6):16–34. doi:10.1109/MSP.2018.2867638.

Doan, M. T., Rajasegarar, S., and Leckie, C. (2015) Profiling Pedestrian Activity Patterns in a Dynamic Urban Environment. In *4th International Workshop on Urban Computing (UrbComp)*, CIKM '15, pages 1827–1830. doi:10.1145/2806416.2806645.

Donoho, D. L. and Grimes, C. (2003) Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. *Proceedings of the National Academy of Sciences*, 100(10):5591–5596. doi:10.1073/pnas.1031596100.

Drucker, H., Burges, C. J., Kaufman, L., Smola, A., and Vapnik, V. (1997) Support Vector Regression Machines. *Advances in neural information processing systems*, 9:155–161. https://proceedings.neurips.cc/paper/1996/file/d38901788c533e8286cb6400b40b386d-Paper.pdf.

Dsilva, C. J., Talmon, R., Gear, C. W., Coifman, R. R., and Kevrekidis, I. G. (2016) Data-Driven Reduction for a Class of Multiscale Fast-Slow Stochastic Dynamical Systems. *SIAM Journal on Applied Dynamical Systems*, 15(3):1327–1351. doi:10.1137/151004896.

Dsilva, C. J., Talmon, R., Coifman, R. R., and Kevrekidis, I. G. (2018) Parsimonious representation of nonlinear dynamical systems through manifold learning: A chemotaxis case study. *Applied and Computational Harmonic Analysis*, 44(3):759–773. doi:10.1016/j.acha.2015.06.008.

Dubois, P. F. (2007) Python: Batteries Included. *Computing in Science &Engineering*, 11. http://csc.ucdavis.edu/~cmg/Group/readings/pythonissue_1of4.pdf.

Eivazi, H., Guastoni, L., Schlatter, P., Azizpour, H., and Vinuesa, R. (2021) Recurrent neural networks and Koopman-based frameworks for temporal predictions in a low-order model of turbulence. *International Journal of Heat and Fluid Flow*, 90:108816. doi:10.1016/j.ijheatfluidflow.2021.108816.

Fernández, Á., Rabin, N., Fishelov, D., and Dorronsoro, J. R. (2020) Auto-adaptive multi-scale Laplacian Pyramids for modeling non-uniform data. *Engineering Applications of Artificial Intelligence*, 93:103682. doi:10.1016/j.engappai.2020.103682.

Folkestad, C., Pastor, D., Mezic, I., Mohr, R., Fonoberova, M., and Burdick, J. (2020) Extended Dynamic Mode Decomposition with Learned Koopman Eigenfunctions for Prediction and Control. In *2020 American Control Conference (ACC)*, pages 3906–3913, Denver, CO, USA. IEEE. doi:10.23919/ACC45564.2020.9147729.

Fowler, M. (2018) *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. ISBN 978-0-13-475768-1. https://learning.oreilly.com/library/view/refactoring-improving-the/9780134757681/.

Froyland, G., Giannakis, D., Lintner, B. R., Pike, M., and Slawinska, J. (2021) Spectral analysis of climate dynamics with operator-theoretic approaches. *Nature Communications*, 12(1):6570. doi:10.1038/s41467-021-26357-x.

García Trillos, N., Gerlach, M., Hein, M., and Slepčev, D. (2020) Error Estimates for Spectral Convergence of the Graph Laplacian on Random Geometric Graphs Toward the Laplace–Beltrami Operator. *Foundations of Computational Mathematics*, 20(4):827–887. doi:10.1007/s10208-019-09436-w.

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., and Wilson, A. G. (2018) GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1809.11165.

Gear, C. W., Kaper, T. J., Kevrekidis, I. G., and Zagaris, A. (2005) Projecting to a Slow Manifold: Singularly Perturbed Systems and Legacy Codes. *SIAM Journal on Applied Dynamical Systems*, 4(3):711–732. doi:10.1137/040608295.

Giannakis, D. (2015) Dynamics-Adapted Cone Kernels. *SIAM Journal on Applied Dynamical Systems*, 14(2):556–608. doi:10.1137/140954544.

Giannakis, D. (2019) Data-driven spectral decomposition and forecasting of ergodic dynamical systems. *Applied and Computational Harmonic Analysis*, 47(2):338–396. doi:10.1016/j.acha.2017.09.001.

Giannakis, D. and Majda, A. J. (2013) Nonlinear Laplacian spectral analysis: capturing intermittent and low-frequency spatiotemporal patterns in high-dimensional data. *Statistical Analysis and Data Mining*, 6(3):180–194. doi:10.1002/sam.11171.

Giannakis, D., Slawinska, J., and Zhao, Z. (2015) Spatiotemporal Feature Extraction with Data-Driven Koopman Operators. In *Feature Extraction: Modern Questions and Challenges*, volume 44, pages 103–115. PMLR. https://proceedings.mlr.press/v44/giannakis15.html.

Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., and Kagal, L. (2018) Explaining Explanations: An Overview of Interpretability of Machine Learning. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 80–89, Turin, Italy. IEEE. doi:10.1109/DSAA.2018.00018.

Goble, C. (2014) Better Research. *IEEE Internet Computing*, 18(5):4–8. doi:10.1109/MIC.2014.88.

Gonzalez, D., Zimmermann, T., and Nagappan, N. (2020) The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 431–442, Seoul Republic of Korea. ACM. doi:10.1145/3379597.3387473.

Gottwald, G. A. and Gugole, F. (2020) Detecting Regime Transitions in Time Series Using Dynamic Mode Decomposition. *Journal of Statistical Physics*, 179(5-6):1028–1045. doi:10.1007/s10955-019-02392-3.

Grad, M. and Raith, T. (2021) Efficient Numerical Solvers for the Manifold Learning Framework datafold. Technical report, Technical University of Munich. https://mediatum.ub.tum.de/doc/1610877/1610877.pdf.

Hafer, L. and Kirkpatrick, A. E. (2009) Assessing open source software as a scholarly contribution. *Communications of the ACM*, 52(12):126–129. doi:10.1145/1610252.1610285.

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., and Wilson, G. (2009) How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. doi:10.1109/SECSE.2009.5069155.

Harlim, J. and Yang, H. (2018) Diffusion Forecasting Model with Basis Functions from QR-Decomposition. *Journal of Nonlinear Science*, 28(3):847–872. doi:10.1007/s00332-017-9430-1.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020) Array programming with NumPy. *Nature*, 585(7825):357–362. doi:10.1038/s41586-020-2649-2.

Helbing, D., Johansson, A., and Al-Abideen, H. Z. (2007) The Dynamics of Crowd Disasters: An Empirical Study. *Physical Review E*, 75(4):046109. doi:10.1103/PhysRevE.75.046109.

Hemati, M. S., Williams, M. O., and Rowley, C. W. (2014) Dynamic mode decomposition for large and streaming datasets. *Physics of Fluids*, 26(11):111701. doi:10.1063/1.4901016.

Hernandez, V., Roman, J. E., Tomas, A., and Vidal, V. (2009) A Survey of Software for Sparse Eigenvalue Problems. Technical report, Universitat Politècnica De València. https://slepc.upv.es.

Herzen, J., Lässig, F., Piazzetta, S. G., Neuer, T., Tafti, L., Raille, G., Van Pottelbergh, T., Pasieka, M., Skrodzki, A., Huguenin, N., Dumonal, M., Kościsz, J., Bader, D., Gusset, F., Benheddi, M., Williamson, C., Kosinski, M., Petrik, M., and Grosch, G. (2021) Darts: User-Friendly Modern Machine Learning for Time Series. *arXiv:2110.03224 [cs, stat]*. https://arxiv.org/abs/2110.03224.

Hinsen, K. (2015) Technical Debt in Computational Science. *Computing in Science & Engineering*, 17(6):103–107. doi:10.1109/MCSE.2015.113.

Hochreiter, S. and Schmidhuber, J. (1997) Long Short-Term Memory. In *Neural Computation*, volume 9 of *8*, pages 1735–1780, 1735–1780. MIT Press. doi:10.1162/neco.1997.9.8.1735.

Hoffmann, M., Scherer, M., Hempel, T., Mardt, A., de Silva, B., Husic, B. E., Klus, S., Wu, H., Kutz, N., Brunton, S. L., and Noé, F. (2021) Deeptime: a Python library for machine learning dynamical models from time series data. *arXiv:2110.15013 [math-ph, physics:physics, stat]*. https://arxiv.org/abs/2110.15013.

Hubbs, C. (2020) A Beginner's Guide to Simulating Dynamical Systems with Python. https://towardsdatascience.com/a-beginners-guide-to-simulating-dynamical-systems-with-python-a29bc27ad9b1. Publication Title: Medium.

Juang, J.-N. and Pappa, R. S. (1985) An eigensystem realization algorithm for modal parameter identification and model reduction. *Journal of Guidance, Control, and Dynamics*, 8(5):620–627. doi:10.2514/3.20031.

Kamb, M., Kaiser, E., Brunton, S. L., and Kutz, J. N. (2020) Time-Delay Observables for Koopman: Theory and Applications. *SIAM Journal on Applied Dynamical Systems*, 19 (2):886–917. doi:10.1137/18M1216572.

Karimi, A. and Georgiou, T. T. (2021) The Challenge of Small Data: Dynamic Mode Decomposition, Redux. *arXiv:2104.04005 [cs, eess, stat]*. https://arxiv.org/abs/2104.04005.

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., and Yang, L. (2021) Physics-informed machine learning. *Nature Reviews Physics*. doi:10.1038/s42254-021-00314-5.

Karunaratne, P., Moshtaghi, M., Karunasekera, S., Harwood, A., and Cohn, T. (2017) Modelling the Working Week for Multi-Step Forecasting using Gaussian Process Regression. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 1994–2000, Melbourne, Australia. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2017/277.

Kemeth, F. P., Haugland, S. W., Dietrich, F., Bertalan, T., Höhlein, K., Li, Q., Bollt, E. M., Talmon, R., Krischer, K., and Kevrekidis, I. G. (2018) An Emergent Space for Distributed Data with Hidden Internal Order through Manifold Learning. *IEEE Access*, 6:77402–77413. doi:10.1109/ACCESS.2018.2882777.

Kevrekidis, I. G. and Samaey, G. (2009) Equation-Free Multiscale Computation: Algorithms and Applications. *Annual Review of Physical Chemistry*, 60(1):321–344. doi:10.1146/annurev.physchem.59.032607.093610.

Kevrekidis, I. G., Gear, C. W., Hyman, J. M., Kevrekidis, G., Runborg, O., and Theodoropoulos, C. (2003) coarse-grained multiscale computation: enabling microscopic simulators to perform system-level analysis. *Communications in Mathematical Sciences*, 4: 715 – 762.

Khomh, F., Adams, B., Cheng, J., Fokaefs, M., and Antoniol, G. (2018) Software Engineering for Machine-Learning Applications: The Road Ahead. *IEEE Software*, 35(5): 81–84. doi:10.1109/MS.2018.3571224.

Kleinmeier, B. (2021) *Modeling of Behavioral Changes in Agent-Based Simulations*. PhD Thesis, Technical University of Munich. https://mediatum.ub.tum.de/?id=1595400.

Kleinmeier, B., Zönnchen, B., Gödel, M., and Köster, G. (2019) Vadere: An Open-Source Simulation Framework to Promote Interdisciplinary Understanding. *Collective Dynamics*, 4:A21. doi:10.17815/CD.2019.21.

Klus, S., Koltai, P., and Schütte, C. (2016) On the numerical approximation of the Perron-Frobenius and Koopman operator. *Journal of Computational Dynamics*, 3(1): 1–12. doi:10.3934/jcd.2016003.

Klus, S., Nüske, F., Peitz, S., Niemann, J.-H., Clementi, C., and Schütte, C. (2020) Data-driven approximation of the Koopman generator: Model reduction, system identification, and control. *Physica D: Nonlinear Phenomena*, 406:132416. doi:10.1016/j.physd.2020.132416.

Koopman, B. O. (1931) Hamiltonian systems and transformation in Hilbert space. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 17(5), pages 315–318. doi:10.1073/pnas.17.5.315.

Korda, M. and Mezić, I. (2018) On Convergence of Extended Dynamic Mode Decomposition to the Koopman Operator. *Journal of Nonlinear Science*, 28(2):687–710. doi:10.1007/s00332-017-9423-0.

Korda, M. and Mezić, I. (2020) Optimal Construction of Koopman Eigenfunctions for Prediction and Control. *IEEE Transactions on Automatic Control*, 65(12):5114–5129. doi:10.1109/TAC.2020.2978039.

Künzner, F. (2020) *Efficient non-intrusive uncertainty quantification for large-scale simulation scenarios*. PhD thesis, Technical University of Munich. https://mediatum.ub.tum.de/?id=1575368.

Kutz, J. N., Brunton, S. L., Brunton, B. W., and Proctor, J. L. (2016) *Dynamic mode decomposition. Data-Driven modelling of complex systems*. Society for Industrial and Applied Mathematics. ISBN 978-1-61197-450-8. https://doi.org/10.1137/1.9781611974508.

Kutz, J. N., Fu, X., and Brunton, S. L. (2016) Multiresolution Dynamic Mode Decomposition. *SIAM Journal on Applied Dynamical Systems*, 15(2):713–735. doi:10.1137/15M1023543.

Kutz, J. N., Proctor, J. L., and Brunton, S. L. (2016) Koopman Theory for Partial Differential Equations. *arXiv:1607.07076 [nlin]*. https://arxiv.org/abs/1607.07076.

Lafon, S., Keller, Y., and Coifman, R. (2006) Data Fusion and Multicue Data Matching by Diffusion Maps. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28 (11):1784–1797. doi:10.1109/TPAMI.2006.223.

Lange, H., Brunton, S. L., and Kutz, J. N. (2021) From Fourier to Koopman: Spectral Methods for Long-term Time Series Prediction. *Journal of Machine Learning Research*, 22(41):1–38. http://jmlr.org/papers/v22/20-406.html.

Längkvist, M., Karlsson, L., and Loutfi, A. (2014) A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42: 11–24. doi:10.1016/j.patrec.2014.01.008.

Layek, G. (2015) *An Introduction to Dynamical Systems and Chaos*. Springer India, New Delhi. doi:10.1007/978-81-322-2556-0.

Le Clainche, S., Vega, J. M., and Soria, J. (2017) Higher order dynamic mode decomposition of noisy experimental data: The flow structure of a zero-net-mass-flux jet. *Experimental Thermal and Fluid Science*, 88:336–353. doi:10.1016/j.expthermflusci.2017.06.011.

LeCun, Y. and Cortes, C. (2010) MNIST handwritten digit database.

LeCun, Y., Bengio, Y., and Hinton, G. (2015) Deep learning. *Nature*, 521(7553):436–444. doi:10.1038/nature14539.

Lee, J. M. (2012) volume 218 of *Graduate Texts in Mathematics*. Springer New York, New York, NY. doi:10.1007/978-1-4419-9982-5.

Lehmberg, D., Dietrich, F., Kevrekidis, I. G., Bungartz, H.-J., and Köster, G. (2020) Exploring Koopman Operator Based Surrogate Models—Accelerating the Analysis of Critical Pedestrian Densities. In *Traffic and Granular Flow 2019*, volume 252, pages 149–157. Springer International Publishing, Cham. doi:10.1007/978-3-030-55973-1_19. Series Title: Springer Proceedings in Physics.

Lehmberg, D., Dietrich, F., Köster, G., and Bungartz, H.-J. (2020) datafold: data-driven models for point clouds and time series on manifolds. *Journal of Open Source Software*, 5(51):2283. doi:10.21105/joss.02283.

Lehmberg, D., Dietrich, F., and Köster, G. (2021) Modeling Melburnians—Using the Koopman operator to gain insight into crowd dynamics. *Transportation Research Part C: Emerging Technologies*, 133:103437. doi:10.1016/j.trc.2021.103437.

Levy, B. (2006) Laplace-Beltrami Eigenfunctions Towards an Algorithm That "Understands" Geometry. In *IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)*, pages 13–13, Matsushima, Japan. IEEE. doi:10.1109/SMI.2006.21.

Li, Q., Dietrich, F., Bollt, E. M., and Kevrekidis, I. G. (2017) Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the Koopman operator. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(10):103111.

Li, W., Wang, J., Fan, R., Zhang, Y., Guo, Q., Siddique, C., and Ban, X. J. (2020) Short-term traffic state prediction from latent structures: Accuracy vs. efficiency. *Transportation Research Part C: Emerging Technologies*, 111:72–90. doi:10.1016/j.trc.2019.12.007.

Lim, B. and Zohren, S. (2021) Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194):20200209. doi:10.1098/rsta.2020.0209.

Lin, B., He, X., and Ye, J. (2015) A geometric viewpoint of manifold learning. *Applied Informatics*, 2(1):3. doi:10.1186/s40535-015-0006-6.

Liu, C., Huang, B., Zhao, M., Sarkar, S., Vaidya, U., and Sharma, A. (2016) Data driven exploration of traffic network system dynamics using high resolution probe data. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 7629–7634, Las Vegas, NV, USA. IEEE. doi:10.1109/CDC.2016.7799448.

Liu, P., Safford, H. R., Couzin, I. D., and Kevrekidis, I. G. (2014) Coarse-grained variables for particle-based models: diffusion maps and animal swarming simulations. *Computational Particle Mechanics*, 1(4):425–440. doi:10.1007/s40571-014-0030-7.

Long, A. W. and Ferguson, A. L. (2019) Landmark Diffusion Maps (L-dMaps): Accelerated manifold learning out-of-sample extension. *Applied and Computational Harmonic Analysis*, 47(1):190–211. doi:10.1016/j.acha.2017.08.004.

Löning, M. and Király, F. (2020) Forecasting with sktime: Designing sktime's New Forecasting API and Applying It to Replicate and Extend the M4 Study. *arXiv:2005.08067 [cs, stat]*. https://arxiv.org/abs/2005.08067.

Lopez, P. A., Wiessner, E., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flotterod, Y.-P., Hilbrich, R., Lucken, L., Rummel, J., and Wagner, P. (2018) Microscopic Traffic Simulation using SUMO. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2575–2582, Maui, HI. IEEE. doi:10.1109/ITSC.2018.8569938.

Lorenz, E. N. (1963) Determinisic Nonperiodic Flow. *Journal of Atmospheric Sciences*, 20.2:130–141. doi:10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.

Lv, Y., Duan, Y., Kang, W., Li, Z., and Wang, F.-Y. (2014) Traffic Flow Prediction With Big Data: A Deep Learning Approach. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–9. doi:10.1109/TITS.2014.2345663.

Ma, W., Chen, L., Zhou, Y., and Xu, B. (2016) What Are the Dominant Projects in the GitHub Python Ecosystem? In *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*, pages 87–95, Wuhan, China. IEEE. doi:10.1109/TSA.2016.23.

Ma, Y. and Fu, Y. (2012) *Manifold learning theory and applications*. CRC ; Taylor & Francis [distributor], Boca Raton, Fla. : London. ISBN 978-1-4398-7109-6.

Majecka, B. (2009) *Statistical models of pedestrian behaviour in the Forum*. PhD Thesis, University of Edinburgh. https://homepages.inf.ed.ac.uk/rbf/FORUMTRACKING/ .

Makridakis, S., Spiliotis, E., and Assimakopoulos, V. (2020)000 time series and 61 forecasting methods. *International Journal of Forecasting*, 36(1):54–74. doi:10.1016/j.ijforecast.2019.04.014.

Manojlović, I., Fonoberova, M., Mohr, R., Andrejčuk, A., Drmač, Z., Kevrekidis, Y., and Mezić, I. (2020) Applications of Koopman Mode Analysis to Neural Networks. *arXiv:2006.11765 [cs, math, stat]*. https://arxiv.org/abs/2006.11765.

Martensen, J. and Rackauckas, C. (2021) DataDrivenDiffEq.jl. https://github.com/SciML/DataDrivenDiffEq.jl. accessed: 01 Dec. 2021.

Matthews, A. G. d. G., van der Wilk, M., Nickson, T., Keisuke, F., Boukouvalas, A., Léon-Villagrá, P., Ghahramani, Z., and Hensman, J. (2017) GPflow: A Gaussian Process Library using TensorFlow. *Journal of Machine Learning Research*, 18(40):1–6.

Mauroy, A. and Goncalves, J. (2020) Koopman-Based Lifting Techniques for Nonlinear Systems Identification. *IEEE Transactions on Automatic Control*, 65(6):2550–2565. doi:10.1109/TAC.2019.2941433.

Mauroy, A. and Mezić, I. (2016) Global Stability Analysis Using the Eigenfunctions of the Koopman Operator. *IEEE Transactions on Automatic Control*, 61(11):3356–3369. doi:10.1109/TAC.2016.2518918.

Mauroy, A., Mezić, I., and Susuki, Y. (2020) *The Koopman Operator in Systems and Control: Concepts, Methodologies, and Applications*, volume 484 of *Lecture Notes in Control and Information Sciences*. Springer International Publishing, Cham. doi:10.1007/978-3-030-35713-9.

McKinney, W. (2011) pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for high performance and scientific computing*, 14(9):1–9.

McQueen, J., Meilă, M., VanderPlas, J., and Zhang, Z. (2016) Megaman: Scalable Manifold Learning in Python. *Journal of Machine Learning Research*, 17(148):1–5. https://jmlr.org/papers/v17/16-109.html.

Melbourne, G. (2021) Pedestrian Counting System - (hourly update frequency), Open Data, Socrata. https://data.melbourne.vic.gov.au/Transport/Pedestrian-Counting-System-Monthly-counts-per-hour/b2ak-trbp.

Melbourne, G. (2021) Pedestrian Counting System - Sensor Locations, Open Data. https://data.melbourne.vic.gov.au/Transport/Pedestrian-Counting-System-Sensor-Locations/h57g-5234.

Mezić, I. (2005) Model Reduction and Decompositions. *Nonlinear Dynamics*, pages 309–325. doi:10.1007/s11071-005-2824-x.

Mezić, I. (2020) and Learning. *arXiv:2010.05377 [math]*. https://arxiv.org/abs/2010.05377.

Mezić, I. and Banaszuk, A. (2004) Comparison of systems with complex behavior. *Physica D: Nonlinear Phenomena*, 197(1-2):101–133. doi:10.1016/j.physd.2004.06.015.

Millman, K. J. and Aivazis, M. (2011) Python for Scientists and Engineers. *Computing in Science & Engineering*, 13(2):9–12. doi:10.1109/MCSE.2011.36.

Muja, M. and Lowe, D. G. (2009) Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *Proceedings of the Fourth International Conference on Computer Vision Theory and Applications*, pages 331–340, Lisboa, Portugal. SciTePress - Science and and Technology Publications. doi:10.5220/0001787803310340.

Muja, M. and Lowe, D. G. (2014) Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240. doi:10.1109/TPAMI.2014.2321376.

Müller, K. R., Smola, A. J., Rätsch, G., Schölkopf, B., Kohlmorgen, J., and Vapnik, V. (1997) Predicting time series with support vector machines. In *Artificial Neural Networks — ICANN'97*, volume 1327, pages 999–1004. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/BFb0020283.

Murphy, K. P. (2012) *Machine learning: a probabilistic perspective*. MIT Press. https://probml.github.io/pml-book/book0.html.

Nadler, B., Lafon, S., Coifman, R. R., and Kevrekidis, I. G. (2006) spectral clustering and reaction coordinates of dynamical systems. *Applied and Computational Harmonic Analysis*, 21(1):113–127. doi:10.1016/j.acha.2005.07.004.

Nagy, A. M. and Simon, V. (2018) Survey on traffic prediction in smart cities. *Pervasive and Mobile Computing*, 50:148–163. doi:10.1016/j.pmcj.2018.07.004.

Nelles, O. (2020) *Nonlinear System Identification: From Classical Approaches to Neural Networks, Fuzzy Models, and Gaussian Processes*. Springer International Publishing, Cham. doi:10.1007/978-3-030-47439-3.

Niemann, J.-H., Klus, S., and Schütte, C. (2021) Data-driven model reduction of agent-based systems using the Koopman generator. *PLOS ONE*, 16(5):e0250970. doi:10.1371/journal.pone.0250970.

Nowogrodzki, A. (2019) How to support open-source software and stay sane. *Nature*, 571(7763):133–134. doi:10.1038/d41586-019-02046-0.

Nyström, E. J. (1930) Über Die Praktische Auflösung von Integralgleichungen mit Anwendungen auf Randwertaufgaben. *Acta Mathematica*, 54(0):185–204. doi:10.1007/BF02547521.

Oliphant, T. E. (2007) Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20. doi:10.1109/MCSE.2007.58.

Olivier, A., Giovanis, D. G., Aakash, B., Chauhan, M., Vandanapu, L., and Shields, M. D. (2020) UQpy: A general purpose Python package and development environment for uncertainty quantification. *Journal of Computational Science*, 47:101204. doi:10.1016/j.jocs.2020.101204.

Otto, S. E. and Rowley, C. W. (2021) Koopman Operators for Estimation and Control of Dynamical Systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1): annurev–control–071020–010108. doi:10.1146/annurev-control-071020-010108.

Papaioannou, P., Talmon, R., di Serafino, D., Kevrekidis, I., and Siettos, C. (2021) Time Series Forecasting Using Manifold Learning. *arXiv:2110.03625 [cs, math]*. https://arxiv.org/abs/2110.03625.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019) High-Performance Deep Learning Library. *Advances in neural information processing systems*, 32:8026–8037. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., and Cournapeau, D. (2011) Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830. https://dl.acm.org/doi/10.5555/1953048.2078195.

Pintelon, R. and Schoukens, J. (2012) *System identification: a frequency domain approach*. John Wiley & Sons Inc, Hoboken, N.J, 2nd ed edition. ISBN 978-0-470-64037-1. https://doi.org/10.1002/0471723134.

Proctor, J. L., Brunton, S. L., and Kutz, J. N. (2016) Dynamic Mode Decomposition with Control. *SIAM Journal on Applied Dynamical Systems*, 15(1):142–161. doi:10.1137/15M1013857.

Proctor, J. L., Brunton, S. L., and Kutz, J. N. (2018) Generalizing Koopman Theory to Allow for Inputs and Control. *SIAM Journal on Applied Dynamical Systems*, 17(1): 909–930. doi:10.1137/16M1062296.

Rabin, N. and Coifman, R. R. (2012) Heterogeneous datasets representation and learning using diffusion maps and Laplacian pyramids. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 189–199. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611972825.17.

Rasmussen, C. E. and Williams, C. K. I. (2006) *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass. ISBN 0-262-18253-X. http://www.gaussianprocess.org/gpml.

Reichstein, M., Camps-Valls, G., Stevens, B., Jung, M., Denzler, J., Carvalhais, N., and Prabhat. (2019) Deep learning and process understanding for data-driven Earth system science. *Nature*, 566(7743):195–204. doi:10.1038/s41586-019-0912-1.

Renggli, C., Karlaš, B., Ding, B., Liu, F., Schawinski, K., Wu, W., and Zhang, C. (2019) Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment. *arXiv:1903.00278 [cs, stat]*. https://arxiv.org/abs/1903.00278.

Rice, J. and Boisvert, R. (1996) From scientific software libraries to problem-solving environments. *IEEE Computational Science and Engineering*, 3(3):44–53. doi:10.1109/99.537091.

Roberts, S., Osborne, M., Ebden, M., Reece, S., Gibson, N., and Aigrain, S. (2013) Gaussian processes for time-series modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1984):20110550. doi:10.1098/rsta.2011.0550.

Roweis, S. T. and Saul, L. K. (2000) Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326. doi:10.1126/science.290.5500.2323.

Rowley, C. W., Mezić, I., Bagheri, S., Schlatter, P., and Henningson, D. S. (2009) Spectral analysis of nonlinear flows. *Journal of Fluid Mechanics*, 641:115–127. doi:10.1017/S0022112009992059.

Runge, J., Bathiany, S., Bollt, E., Camps-Valls, G., Coumou, D., Deyle, E., Glymour, C., Kretschmer, M., Mahecha, M. D., Muñoz-Marí, J., van Nes, E. H., Peters, J., Quax, R., Reichstein, M., Scheffer, M., Schölkopf, B., Spirtes, P., Sugihara, G., Sun, J., Zhang, K., and Zscheischler, J. (2019) Inferring causation from time series in Earth system sciences. *Nature Communications*, 10(1):2553. doi:10.1038/s41467-019-10105-3.

Saltelli, A., Bammer, G., Bruno, I., Charters, E., Di Fiore, M., Didier, E., Nelson Espeland, W., Kay, J., Lo Piano, S., Mayo, D., Pielke Jr, R., Portaluri, T., Porter, T. M., Puy, A., Rafols, I., Ravetz, J. R., Reinert, E., Sarewitz, D., Stark, P. B., Stirling, A., van der Sluijs, J., and Vineis, P. (2020) Five ways to ensure that models serve society: a manifesto. *Nature*, 582(7813):482–484. doi:10.1038/d41586-020-01812-9.

Sauer, T., Yorke, J. A., and Casdagli, M. (1991) Embedology. *Journal of statistical Physics*, 65(3):579–616. doi:10.1007/BF01053745.

Schmid, P. J. (2010) Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28. doi:10.1017/S0022112010001217.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015) Hidden Technical Debt in Machine Learning Systems. In *Advances in neural information processing systems*, volume 28, pages 2503–2511. https://dl.acm.org/doi/10.5555/2969442.2969519.

Seabold, S. and Perktold, J. (2010) Statsmodels: Econometric and Statistical Modeling with Python. In *Proceedings of the 9th Python in Science Conference*, pages 92–96, Austin, Texas. doi:10.25080/Majora-92bf1922-011.

Seitz, M. J. and Köster, G. (2012) Natural discretization of pedestrian movement in continuous space. *Physical Review E*, 86(4):046108. doi:10.1103/PhysRevE.86.046108.

Shen, C. and Wu, H.-T. (2020) Scalability and robustness of spectral embedding: landmark diffusion is all you need. *arXiv:2001.00801 [math, stat]*. https://arxiv.org/abs/2001.00801.

Smaragdakis, Y. and Batory, D. (2002) Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255. doi:10.1145/505145.505148.

Smith, A. M., Niemeyer, K. E., Katz, D. S., Barba, L. A., Githinji, G., Gymrek, M., Huff, K. D., Madan, C. R., Mayes, A. C., Moerman, K. M., Prins, P., Ram, K., Rokem, A., Teal, T. K., Guimera, R. V., and Vanderplas, J. T. (2018) Journal of Open Source Software (JOSS): design and first-year review. *PeerJ Computer Science*, 4:e147. doi:10.7717/peerj-cs.147.

Snoek, J., Larochelle, H., and Adams, R. P. (2012) Practical Bayesian Optimization of Machine Learning Algorithms. In *Practical bayesian optimization of machine learning algorithms*, volume 25 of *NIPS'12*, pages 2951–2959. https://dl.acm.org/doi/10.5555/2999325.2999464.

Sonnenburg, S., Braun, M. L., Ong, C. S., Bengio, S., Bottou, L., Holmes, G., LeCun, Y., Müller, K.-R., Pereira, F., Rasmussen, C. E., Rätsch, G., Schölkopf, B., Smola, A., Vincent, P., Weston, J., and Williamson, R. (2007) The Need for Open Source Software in Machine Learning. *Journal of Machine Learning Research*, 8(81):2443–2466. https://jmlr.org/papers/v8/sonnenburg07a.html.

Stodden, V. and Miguez, S. (2014) Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research. *Journal of Open Research Software*, 2(1):e21. doi:10.5334/jors.ay.

Strange, H. and Zwiggelaar, R. (2014) *Open Problems in Spectral Dimensionality Reduction*. SpringerBriefs in Computer Science. Springer International Publishing, Cham. doi:10.1007/978-3-319-03943-5.

Strogatz, S. H. (2015) *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering*. Westview Press, a member of the Perseus Books Group, Boulder, CO, second edition edition. ISBN 978-0-8133-4910-7.

Surana, A. (2020) Koopman Operator Framework for Time Series Modeling and Analysis. *Journal of Nonlinear Science*, 30(5):1973–2006. doi:10.1007/s00332-017-9441-y.

Takeishi, N., Kawahara, Y., Tabei, Y., and Yairi, T. (2017) Bayesian Dynamic Mode Decomposition. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 2814–2821, Melbourne, Australia. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2017/392.

Takens, F. (1981) Detecting strange attractors in turbulence. In *Dynamical Systems and Turbulence, Warwick 1980*, volume 898, pages 366–381. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/BFb0091924.

Tangirala, A. K. (2018) *Principles of system identification: theory and practice*. CRC Press. ISBN 978-1-315-22250-9. https://doi.org/10.1201/9781315222509.

Tavenard, R., Faouzi, J., Vandewiele, G., Divo, F., Androz, G., Holtz, C., Payne, M., Yurchak, R., Rußwurm, M., Kolar, K., and Woods, E. (2020) A Machine Learning Toolkit for Time Series Data. *Journal of Machine Learning Research*, 21(118):1–6. https://jmlr.org/papers/v21/20-091.html.

Thiede, E. H., Trstanova, Z., and Banisch, R. (2021) pyDiffMap. https://github.com/DiffusionMapsAcademics/pyDiffMap.

Thompson, N. C., Greenewald, K., Lee, K., and Manso, G. F. (2020) The Computational Limits of Deep Learning. *arXiv:2007.05558 [cs, stat]*. https://arxiv.org/abs/2007.05558.

Tu, J. H., Rowley, C. W., Luchtenburg, D. M., Brunton, S. L., and Kutz, J. N. (2014) On Dynamic Mode Decomposition: Theory and Applications. *Journal of Computational Dynamics*, 1(2):391–421. doi:10.3934/jcd.2014.1.391.

Umlauft, J., Beckers, T., and Hirche, S. (2018) Scenario-based Optimal Control for Gaussian Process State Space Models. In *2018 European Control Conference (ECC)*, pages 1386–1392, Limassol. IEEE. doi:10.23919/ECC.2018.8550458.

Upadhyay, K., Giovanis, D. G., Alshareef, A., Johnson, C. L., Carass, A., Bayly, P. V., and Ramesh, K. T. (2021) Data-driven Uncertainty Quantification in Computational Human Head Models. *arXiv:2110.15553*, page 27. https://arxiv.org/abs/2110.15553v1.

VanRossum, G. and Drake, F. L. (2010) The Python language reference. Technical report, Python Software Foundation. https://docs.python.org/3.6/reference/.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., and van Mulbregt, P. (2020) SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272. doi:10.1038/s41592-019-0686-2.

Vlahogianni, E. I., Karlaftis, M. G., and Golias, J. C. (2014) Short-term traffic forecasting: Where we are and where we're going. *Transportation Research Part C: Emerging Technologies*, 43:3–19. doi:10.1016/j.trc.2014.01.005.

von Neumann, J. (1932) Proof of the quasi-ergodic hypothesis. *Proceedings of the National Academy of Sciences*, 18(1):70–82. doi:10.1073/pnas.18.1.70.

von Sivers, I. and Köster, G. (2015) Dynamic stride length adaptation according to utility and personal space. *Transportation Research Part B: Methodological*, 74:104–117. doi:10.1016/j.trb.2015.01.009.

von Sivers, I., Templeton, A., Künzner, F., Köster, G., Drury, J., Philippides, A., Neckel, T., and Bungartz, H.-J. (2016) Modelling social identification and helping in evacuation simulation. *Safety Science*, 89:288–300. doi:10.1016/j.ssci.2016.07.001.

Wang, X., Mcintosh, W., Liono, J., and Salim, F. D. (2018) Predicting the city foot traffic with pedestrian sensor data. In *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, Melbourne, Australia. ACM. doi:10.4108/eai.7-11-2017.2273699.

Wiese, I., Polato, I., and Pinto, G. (2020) Naming the Pain in Developing Scientific Software. *IEEE Software*, 37(4):75–82. doi:10.1109/MS.2019.2899838.

Williams, M. O., Rowley, C. W., and Kevrekidis, I. G. (2014) A Kernel-Based Approach to Data-Driven Koopman Spectral Analysis. *arXiv:1411.2260 [math]*. https://arxiv.org/abs/1411.2260.

Williams, M. O., Kevrekidis, I. G., and Rowley, C. W. (2015) A Data–Driven Approximation of the Koopman Operator: Extending Dynamic Mode Decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346. doi:10.1007/s00332-015-9258-5.

Williams, M. O., Hemati, M. S., Dawson, S. T., Kevrekidis, I. G., and Rowley, C. W. (2016) Extending Data-Driven Koopman Analysis to Actuated Systems. *IFAC-PapersOnLine*, 49(18):704–709. doi:10.1016/j.ifacol.2016.10.248.

Xiao, H., Rasul, K., and Vollgraf, R. (2017) Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv:1708.07747 [cs, stat]*. https://arxiv.org/abs/1708.07747.

Yao, H., Tang, X., Wei, H., Zheng, G., and Li, Z. (2019) Revisiting Spatial-Temporal Similarity: A Deep Learning Framework for Traffic Prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:5668–5675. doi:10.1609/aaai.v33i01.33015668.

Yin, X., Wu, G., Wei, J., Shen, Y., Qi, H., and Yin, B. (2020) A Comprehensive Survey on Traffic Prediction. *arXiv:2004.08555 [cs, eess]*. https://arxiv.org/abs/2004.08555.

Yokoyama, H. (2019) Machine Learning System Architectural Pattern for Improving Operational Stability. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 267–274, Hamburg, Germany. IEEE. doi:10.1109/ICSA-C.2019.00055.

Zameni, M., He, M., Moshtaghi, M., Ghafoori, Z., Leckie, C., Bezdek, J. C., and Ramamohanarao, K. (2019) Urban Sensing for Anomalous Event Detection: Distinguishing Between Legitimate Traffic Changes and Abnormal Traffic Variability. In *Machine Learning and Knowledge Discovery in Databases*, volume 11053, pages 553–568. Springer International Publishing, Cham. doi:10.1007/978-3-030-10997-4_34.

Zhang, H., Rowley, C. W., Deem, E. A., and Cattafesta, L. N. (2019) Online Dynamic Mode Decomposition for Time-Varying Systems. *SIAM Journal on Applied Dynamical Systems*, 18(3):1586–1609. doi:10.1137/18M1192329.

Zönnchen, B. (2021) *Effcient parallel algorithms for large-scale pedestrian simulation*. PhD Thesis, Technical University of Munich. https://mediatum.ub.tum.de/node?id=1593965.

Supplementary material

The supplementary material comprises the software that I developed within my thesis as well as the data that I used to obtain my results. All files are hosted and archived in an online repository managed by "Zenodo"[1].

Download supplementary material (76.5 MB, zipped)

https://doi.org/10.5281/zenodo.5748366

The repository has the following folder structure:

- **`datafold_1.1.6/`**: Software *datafold* version 1.1.6

  See `README.rst` for instructions on how to install the software. For the current development process of *datafold* see also:

  * Repository: https://gitlab.com/datafold-dev/datafold
  * Documentation: https://datafold-dev.gitlab.io/datafold/

- **`rdist/`**: Software to efficiently compute a sparse distance matrix. See `README.rst` for instructions on how to compile and install *mlpack* as part of *rdist*. Note that the data used for the benchmark analysis is not included because it is too large.

- **`scripts/`**: In each folder the Python scripts that are intended to be executed have the prefix "`main_`":

  – `01example_ode/`: Generate example dynamical system in Fig. 2.1
  – `02swiss_roll_manifold/`: Generate swiss-roll example in Fig. 2.5, Fig. 2.6, Fig. 2.7, and Fig. 3.10
  – `03package_download_stats/`: Obtain the download statistics in Fig. 2.10
  – `04pendulum/`: Reproduce results of Section 4.1
  – `05bus_station/`: simulator *Vadere*, data and scripts to reproduce results of Section 4.2
  – `06melbourne_pedestrian_counting/`: Data and scripts to reproduce results from Section 4.3
  – `07benchmark_rdist`: Benchmark results and scripts to generate figures and table in Section 4.4

[1] Zenodo (https://zenodo.org/) is an open repository to share all kinds of scientific output.