©Copyright 2020
Brian de Silva

Data-driven discovery and model reduction of complex systems

Brian de Silva

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

J. Nathan Kutz, Chair

Steven L. Brunton

Aleksandr Aravkin

Program Authorized to Offer Degree:
Department of Applied Mathematics

University of Washington

Abstract

Data-driven discovery and model reduction of complex systems

Brian de Silva

Chair of the Supervisory Committee:
Robert Bolles and Yasuko Endo Professor J. Nathan Kutz
Applied Mathematics

Dynamical systems play an integral role in the continued success of scientific theories in describing and predicting the world around us. They are at the heart of countless scientific models, including electromagnetic theory (Maxwell's equations and Lorenz's force law), general relativity (the Einstein field equations), and quantum mechanics (the Schrödinger equation). This thesis considers three problems related to the practical use of dynamical systems in scientific inquiry: *identifying* a system, leveraging *predictions* from dynamical systems in applications, and efficiently *solving* a class of dynamical systems. We begin by applying the sparse identification of nonlinear dynamical systems (SINDy) technique for *system identification* to the canonical problem of falling bodies. Using an experimental dataset consisting of noisy measurements of sports balls dropped from a tall bridge, we highlight challenges faced by practitioners attempting to perform system identification when working with real-world datasets and provide strategies for overcoming them. We briefly present PySINDy, an open source software package in Python designed for the sparse identification of nonlinear dynamical systems. Next we turn to the problem of detecting sensor faults in airplane flight data. We propose a simple physics-based approach which learns a model for the underlying relationships between sensors. Online, the method *predicts* future sensor readings from current ones, flagging potential sensor failures when incoming measurements disagree with predictions. The overall model is a hybrid of ideas and methods from dynamic

mode decomposition, Kalman filters, and machine learning. Its performance is demonstrated on two artificial, but realistic, flight datasets, and one real-world one. Finally, we introduce a numerical method for the efficient *solution* of parametrized elliptic partial differential equations (PDEs) on component-based domains. The method takes advantage of analytical properties of the differential operator defining the PDE, the component-based nature of the domain, and empirically-discovered structure in solutions of the PDE. The result is a compact reduced-order model which can be efficiently evaluated to approximate solutions to the PDE in a many-query setting.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisors, Nathan and Steve. They took me on as a student during a difficult time in my academic career and for that I am forever grateful. But beyond that they were also a delight to work with. They were a constant source of encouragement, their guidance was indispensable in helping me mature as a researcher, and they always made me feel like they were in my corner. I would also especially like to thank Nathan for the countless vegan lattes he made for me during our meetings.

I also owe my thanks to several other collaborators and mentors. I am thankful to Greg for staying on my committee despite setbacks in my academic progress. I want to thank Sasha for being willing to go through my code line-by-line with me and for showing me firsthand how excited and enthusiastic someone can be about math. I am indebted to Ulrich for investing so much effort in my early development as a mathematician. I feel lucky to have had the opportunity to work with an exceptional group of collaborators; Kathleen, Jared, Jonathan, and Markus have all contributed to my appreciation for how rewarding and enjoyable collaborative projects can be. I would also like to extend my gratitude to Lauren Lederer for her assistance in navigating the logistical side of graduate school.

Thanks to my friends Jeremy, Andreas, Kathleen, Alex, and Weston, who were there from the beginning of this journey, for helping to make graduate school one of the best periods of my life. Thanks to Emily and Ellie for their support and for talking to me about topics other than math. Thanks most of all to my parents for their constant encouragement throughout my 24 years of schooling.

Finally, I would like to thank two organizations within the Applied Mathematics department: the diversity committee—for enriching my graduate school experience with an

added sense of purpose—and the numerical analysis research club (NARC)—for providing stimulating problems to ponder that were separate from my research.

DEDICATION

to my parents, Sarah and Amal

Chapter 1

INTRODUCTION

Since its inception the scientific process has been remarkably successful at explaining physical phenomena. From planetary motion down to quantum mechanics, from ecology down to cellular biology, from climate science down to fluid dynamics, scientific theories have made vast contributions to our understanding of nearly every aspect of the natural world. A hallmark of scientific theories is their ability to produce accurate predictions. Neptune was discovered after it was observed that the positions of Uranus' orbit differed from those Newton's laws of motion and gravitation predicted. Many scientific theories capable of making quantitative predictions have, as one of their core components, a model, often in the form of a set of governing equations. Maxwell's equations, along with Lorenz's force law are the basis for classical electromagnetic theory; the Einstein field equations lie at the heart of general relativity, which itself supplanted Newton's law of universal gravitation; and the Navier-Stokes equations provide a mathematical model of viscous fluid flow. Wherever one looks in the natural sciences, one is sure to find laws, equations, or quantitative models of some sort. One of the most widespread and accomplished classes of such models are dynamical systems: systems of differential equations that govern the evolution of a set of variables in time. Part of what makes dynamical systems so powerful is that they often allow one to extrapolate to new sets of initial conditions, a task at which machine learning algorithms tend to struggle.

Once identified, the natural next step is to attempt to *solve* the dynamical system to make *predictions* about the behavior of the system of interest under different conditions. Given measurements from earthquake occurring in Japan, for example, one might use a fluid

dynamics model to evolve a resulting ocean wave forward in time and predict which areas along the west coast of the United States are most likely to be struck with a tsunami. Note that this forecast is made possible by both a fluid model and a method of quickly obtaining solutions to the model. A simulation taking longer to complete than the wave takes to reach the west coast is of little use in this instance.

Historically, governing equations or laws were derived by hand; either from first principles or by meticulously extracting patterns from measurement data. For example, Kepler famously formulated his first two laws of planetary motion after poring over Brahe's measurements of planetary motion. More recently data-driven methods for extracting physical laws from datasets have begun occupying increasingly prominent positions in scientists' toolkits. Such techniques boast the ability to allow scientists to extract simple, interpretable rules from huge, complicated datasets. They are often capable of detecting nuanced or hidden relationships between variables that would be impossible for a human to pick out (simply due to the staggering size of modern datasets). These methods can viewed as complementary to analytical techniques: once posited, dynamical systems and their solutions lend themselves to myriad tools for theoretical study.

After governing equations are known, they can be leveraged to make predictions about the world, assuming they can be solved. Model forecasts are use to anticipate things like the paths of hurricanes, the spread of infectious diseases, how chemicals will react with one another, the weather, how pollutants will dissipate in different mediums, which protein structures will prove useful for some task, or the proliferation of invasive species. They can also be used to detect when a system starts to behave in unexpected ways, enabling the automatic monitoring of systems of interest for anomalous activity.

An already sizable and ever-growing set of techniques has been proposed for solving dynamical systems. Initially the steps of these algorithms were carried out by hand, but in the modern era these calculations are typically delegated to computers, enabling problems of increasing size and complexity to be tackled. Numerical methods can be separated into two broad categories: general purpose and specialized methods. General purpose algorithms,

such as Runge-Kutta methods in numerical analysis or gradient descent in optimization, tend to be suitable for large classes of problems. They are easy to apply, but may not give the best performance. Specialized techniques, on the other hand, are as their name suggests; specialized to particular types of equations or domains. They often exploit theoretical properties of the equations they attempt to solve in order to improve accuracy, speed, or both. Typically the more that is known about the equations, the more performance gains one can expect to achieve. General methods are often preferable if one needs to solve a problem only a small number of times. A specialized method shines when the problem must be solved many times, or if other properties of the problem make it difficult for a general method to handle. It is not uncommon to encounter an application in which hundreds, thousands, or millions of conditions need to be modeled.

In this thesis we touch on three problems related to mathematical models: discovering them, efficiently solving them, and leveraging their predictions. We work with both artificially generated and experimentally collected datasets, using the artificial datasets to validate our methods and the experimental ones to demonstrate that they work in real-world applications.

1.1 Organization

In Chapter 2 we take a data-driven approach to a classical problem in mechanics: falling bodies. Equipped with noisy measurement data of various sports balls dropped from a tall bridge, we use the Sparse Identification of Nonlinear Dynamical systems method (SINDy) to identify viable dynamical systems to model the balls' trajectories. A group sparsity regularization is used to temper the effects of noise on the model discovery process, finding that this regularized version of SINDy is able to detect constant gravitational acceleration and a drag term that is linear in velocity. We use this simple problem to highlight some of the issues that can arise when machine learning methods are used for system identification and suggest strategies for overcoming such obstacles.

Chapter 3 introduces PySINDy, an open source software package in Python for the discov-

ery of nonlinear dynamical systems from data. This package was written to be the de facto standard implementation of the SINDy algorithm in Python and will hopefully empower other scientists with the ability to quickly and easily bring model discovery to their research. This chapter discusses the rationale behind design decisions and provides an overview of features implemented in PySINDy along with examples.

In Chapter 4 we propose a simple physics-based approach for sensor fault detection in airplane flight tests. The goal in this application is to determine when a sensor or group of sensors has stopped giving accurate readings. Given "healthy" measurements from an array of sensors, a linear time-invariant (LTI) model for the interactions between sensors is learned using the dynamic mode decomposition for control (DMDc) method with time-delays. Having learned the interactions between sensors, this model is then used to predict future states of sensor outputs, given current measurements. A sensor fault is presumed to have occurred when readings from the next time step disagree with their predicted values by too large a margin. We benchmark our method on two simulated, but realistic, flight-related datasets and one real-world flight test dataset. The method exhibits satisfactory performance in all cases.

Chapter 5 concerns the efficient solution of a *known* differential equation. We introduce a method suitable for a parametrized elliptic partial differential equation (PDE) on a component-based domain which needs to be solved numerous times with different parameter values. Our method is an extension of one introduced by Eftang and Patera [37] in a more general case when different components of the domain intersect at corners, leading to singularities in the solution of the PDE for some parameter configurations. The algorithm exploits analytic properties of the parametrized PDE, the component-based layout of the domain, and structure empirically discovered in numerically computed solutions to achieve considerable computational speedups.

Chapter 2

DISCOVERY OF PHYSICS FROM DATA: DISAMBIGUATING FLUID FORCES ON FALLING BODIES

This chapter is based on joint work with David Higdon, Steve Brunton, and Nathan Kutz [35].

2.1 Introduction

The ability to derive governing equations and physical principles has been a hallmark feature of scientific discovery and technological progress throughout human history. Even before the scientific revolution, the Ptolemaic doctrine of the *perfect circle* [86, 84] provided a principled decomposition of planetary motion into a hierarchy of circles, i.e. a bona fide theory for planetary motion. The scientific revolution and the resulting development of calculus provided the mathematical framework and language to precisely describe scientific principles, including gravitation, fluid dynamics, electromagnetism, quantum mechanics, etc. With advances in data science over the past few decades, principled methods are emerging for such scientific discovery from time-series measurements alone. Indeed, across the engineering, physical and biological sciences, significant advances in sensor and measurement technologies have afforded unprecedented new opportunities for scientific exploration. Despite its rapid advancements and wide-spread deployment, *machine learning* (ML) and *artificial intelligence* (AI) algorithms for scientific discovery face significant challenges and limitations, including noisy and corrupt data, latent variables, multiscale physics, and the tendency for overfitting. In this chapter, we revisit one of the classic problems of physics considered by Galileo and Newton, that of falling objects and gravitation. We demonstrate that a sparse regression framework is well-suited for physics discovery, while highlighting both the need for principled

methods to extract parsimonious physics models and the challenges associated with the naive application of ML/AI techniques. Even this simplest of physical examples demonstrates critical principles that must be considered in order to make data-driven discovery viable across the sciences.

Measurements have long provided the basis for the discovery of governing equations. Through empirical observations of planetary motion, the Ptolemaic theory of motion was developed [86, 84]. This was followed by Kepler's laws of planetary motion and the elliptical courses of planets in a heliocentric coordinate system [55]. By hand calculation, he was able to regress Brahe's state-of-the-art data on planetary motion to the minimally parametrized elliptical orbits which described planetary orbits with a terseness the Ptolemaic system had never managed to achieve. Such models led to the development of Newton's $\mathbf{F} = m\mathbf{a}$ [80], which provided a universal, generalizable, interpretable, and succinct description of physical dynamics. Parsimonious models are critical in the philosophy of Occam's razor: the simplest set of explanatory variables is often the best [12, 36, 13, 104]. It is through such models that many technological and scientific advancements have been made or envisioned. To highlight this point, consider any textbook in the sciences and note that the vast majority of governing equations offered on any subject have only a small number of terms to model the dynamics.

What is largely unacknowledged in the scientific discovery process is the intuitive leap required to formulate physics principles and governing equations. Consider the example of falling objects. According to physics folklore, Galileo discovered, through experimentation, that objects fall with the same constant acceleration, thus disproving Aristotle's theory of gravity, which stated that objects fall at different speeds depending on their mass. The leaning tower of Pisa is often the setting for this famous stunt, although there is little evidence such an experiment actually took place [32, 3, 106]. Indeed, many historians consider it to have been a thought experiment rather than an actual physical test. Many of us have been to the top of the leaning tower and have longed to drop a bowling ball from the top, perhaps along with a golf ball and soccer ball, in order to replicate this experiment. If we were to perform such a test, here is what we would likely find: Aristotle was correct. Balls of differ-

ent masses and sizes *do* reach the ground at different times. As we will show from our own data on falling objects, (noisy) experimental measurements may be insufficient for discovering a constant gravitational acceleration, especially when the objects experience Reynolds numbers varying by orders of magnitudes over the course of their trajectories. But what is beyond dispute is that Galileo did indeed *posit* the idea of a fixed acceleration, a conclusion that would have been exceptionally difficult to come to from such measurement data alone. Gravitation is only one example of the intuitive leap required for a paradigm shifting physics discovery. Maxwell's equations [69] have a similar story arc revolving around Coulomb's inverse square law. Maxwell cited Coulomb's torsion balance experiment as establishing the inverse square law while dismissing it only a few pages later as an approximation [39, 8]. Maxwell concluded that Faraday's observation that an electrified body, touched to the inside of a conducting vessel, transfers all its electricity to the outside surface as much more direct proof of the square law. In the end, both would have been approximations, with Maxwell taking the intuitive leap that exactly a power of negative two was needed when formulating Maxwell's equations. Such examples abound across the sciences, where intuitive leaps are made and seminal theories result.

One challenge facing ML and AI methods is their inability to take such leaps. At their core, many ML and AI algorithms involve regressions based on data, and are statistical in nature [16, 132, 11, 76]. Thus by construction, a model based on measurement data would not produce an exact inverse *square* law, but rather a slightly different estimate of the exponent. In the case of falling objects, ML and AI would yield an Aristotelian theory of gravitation, whereby the data would suggest that objects fall at a speed related to their mass. As we will show, the trajectories of such falling objects are quite similar, although the differences are statistically sufficient to support such a hypothesis. Of course, even Galileo intuitively understood that air resistance plays a significant role in the physics of falling objects, which is likely the reason he conducted controlled experiments on inclined ramps. Although we understand that air resistance, which is governed by latent fluid dynamic variables, explains the discrepancy between the data and a constant gravity model, our algorithms do not. Without

modeling these small disparities (e.g., due to friction, heat dissipation, air resistance, etc.), it is almost impossible to uncover universal laws such as gravitation. Differences between theory and data have played a foundational role in physics, with general relativity arising from inconsistencies between gravitational theory and observations, and quantum mechanics arising from our inability to explain the photoelectric effect with Maxwell's equations.

Our goal in this chapter is to highlight the many subtle and nuanced concerns related to data-driven discovery using modern ML and AI methods. Specifically, we highlight these issues on the most elementary of problems: modeling the motion of falling objects. Given our ground-truth knowledge of the physics, this example provides a convenient testbed for different physics discovery techniques. It is important that one clearly understands the potential pitfalls in such methods before applying them to more sophisticated problems which may arise in fields like biology, neuroscience, and climate modeling. Notably, it serves as a cautionary tale about what can actually be concluded about physics principles and governing equations from measurement data. Our physics discovery method is rooted in the *sparse identification for nonlinear dynamics* (SINDy) algorithm, which has been shown to extract parsimonious governing equations in a broad range of physical sciences [21]. SINDy has been widely applied to identify models for fluid flows [62, 63], optical systems [109], chemical reaction dynamics [47], convection in a plasma [34], structural modeling [58], and for model predictive control [53]. There are also a number of theoretical extensions to the SINDy framework, including for identifying partial differential equations [96, 100], and models with rational function nonlinearities [67]. It can also incorporate partially known physics and constraints [62]. The algorithm can be reformulated to include integral terms for noisy data [101] or handle incomplete or limited data [120, 102]. These diverse mathematical developments provide a mature framework for broadening the applicability of the model discovery method. In this chapter we show that *group sparsity* [95] may be used to enforce that the same model terms explain *all* of the observed trajectories, which is essential in identifying the correct model terms without overfitting.

SINDy is by no means the only attempt that has been made at using machine learn-

ing to infer physical models from data. Gaussian processes have been employed to learn conservation laws described by parametric linear equations [89]. Symbolic regression has been successfully applied to the problem of inferring dynamics from data [13, 104]. Another closely related set of approaches are process-based models [116, 19, 117] which, similarly to SINDy, allow one to specify a library of relationships or functions between variables based on domain knowledge and produce an interpretable set of governing equations. The principal difference between process-based models and SINDy is that SINDy employs sparse regression techniques to perform function selection which allows a larger class of library functions to be considered than is tractable for process-based models. Deep learning methods have been proposed for accomplishing a variety of related tasks such as predicting physical dynamics directly [75], building neural networks that respect given physical laws [90], discovering parameters in nonlinear partial differential equations with limited measurement data [91], and simultaneously approximating the solution and nonlinear dynamics of nonlinear partial differential equations [88]. Graph neural networks [10], a specialized class of neural networks that operate on graphs, have been shown to be effective at learning basic physics simulators from measurement data [9, 29] and directly from videos [127]. It should be noted that the aforementioned neural network approaches either require detailed prior knowledge of the form of the underlying differential equations or fail to yield simple sets of interpretable governing equations.

2.2 Materials

2.2.1 Fluid forces on a sphere: A brief history

It must have been immediately clear to Galileo and Newton that committing to a gravitational constant created an inconsistency with experimental data. Specifically, one had to explain why objects of different sizes and shapes fall at different speeds (e.g. a feather versus a cannon ball). Wind resistance was an immediate candidate to explain the *discrepancy* between a universal gravitational constant and measurement data. The fact that Galileo per-

formed experiments where he rolled balls down inclines seems to suggest that he was keenly aware of the need to isolate and disambiguate the effects of gravitational forces from fluid drag forces. Discrepancies between the Newtonian theory of gravitation and observational data of Mercury's orbit led to Einstein's development of general relativity. Similarly, the photoelectric effect was a discrepancy in Maxwell's equations which led to the development of quantum mechanics.

Discrepancy modeling is therefore a critical aspect of building and discovering physical models. Consider the motion of falling spheres as a prototypical example. In addition to the force of gravity, a falling sphere encounters a fluid drag force as it passes through the air. A simple model of the drag force F_D is given by:

$$F_D = \frac{1}{2}\rho v^2 A C_D, \tag{2.1}$$

where ρ is the fluid density, v is the velocity of the sphere with respect to the fluid, $A = \pi D^2/4$ is the cross-sectional area of the sphere, D is the diameter of the sphere, and C_D is the dimensionless drag coefficient. As the sphere accelerates through the fluid, its velocity increases, exciting various unsteady aerodynamic effects, such as laminar boundary layer separation, vortex shedding, and eventually a turbulent boundary layer and wake [74, 66, 26, 1, 2, 108]. Thus, the drag coefficient is a function of the sphere's velocity, and this coefficient generally decreases for increasing velocity. Figure 2.1 shows the drag coefficient C_D for a sphere as a function of the Reynolds number $Re = \rho v D/\mu$, where μ is the dynamic viscosity of the fluid; for a constant diameter and viscosity, the Reynolds number is directly proportional to the velocity. Note that the drag coefficient of a smooth sphere will differ from that of a rough sphere. The flow over a rough sphere will become turbulent at lower velocities, causing less flow separation and a more streamlined, lower-drag wake; this explains why golf balls are dimpled, so that they will travel farther [108]. Thus, (2.1) states that drag is related to the square of the velocity, although C_D has a weak dependence on velocity. When Re is small, C_D is proportional to $1/v$, resulting in a drag force that is linear in v. For larger values of Re, C_D is approximately constant (away from the steep drop), leading to a

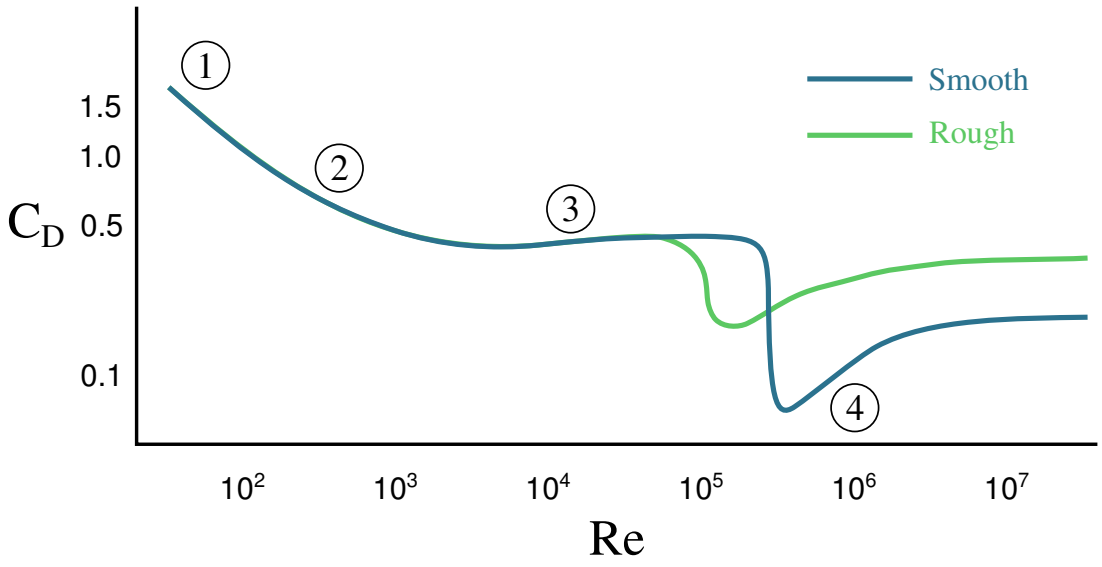


Figure 2.1: The drag coefficient for a sphere as a function of Reynolds number, *Re*. The dark curve shows the coefficient for a sphere with a smooth surface and the light curve a sphere with a rough surface. The numbers highlight different flow regimes. (1) attached flow and steady separated flow; (2) separated unsteady flow, with laminar flow boundary layer upstream of separation, producing a Kármán vortex street; (3) separated unsteady flow with a chaotic turbulent wake downstream and a laminar boundary layer upstream; (4) post-critical separated flow with turbulent boundary layer.

quadratic drag force. Eventually, the drag force will balance the force of gravity, resulting in the sphere reaching its *terminal velocity*. In addition, as the fluid wake becomes unsteady, the drag force will also vary in time, although these variations are typically fast and may be time-averaged. Finally, objects accelerating in a fluid will also accelerate the fluid out of the way, resulting in an effective mass that includes the mass of the body and an *added mass* of accelerated fluid [79]; however, this added mass force will typically be quite small in air.

In addition to the theoretical study of fluid forces on an idealized sphere, there is a rich history of scientific inquiry into the aerodynamics of sports balls [108, 42, 72, 71]. Apart from gravity and drag, a ball's trajectory can be influenced by the spin of the ball via the Magnus force or lift force which acts in a direction orthogonal to the drag. Other factors that

can affect the forces experienced by a falling ball include air temperature, wind, elevation, and ball surface shape.

2.2.2 Data set

The data considered in this chapter are height measurements of balls falling through air. These measurements originate from two sources: physical experiments and simulations. Such experiments are popular in undergraduate physics classes where they are used to explore linear versus quadratic drag [82, 52, 33, 31] and scaling laws [114]. In June 2013 a collection of balls, pictured in Figure 2.2, were dropped, *twice each*, from the Alex Fraser Bridge in Vancouver, BC from a height of about 35 meters above the landing site. In total 11 balls were dropped: a golf ball, a baseball, two whiffle balls with elongated holes, two whiffle balls with circular holes, two basketballs, a bowling ball, and a volleyball (not pictured). More information about the balls is given in Table 2.1. The air temperature at the time of the drops was 65 degrees Fahrenheit (18° Celsius). A hand held iPad was used to record video of the drops at a rate of 15 frames per second. The height of the falling objects was then estimated by tracking the balls in the resulting videos. Figure 2.3 visualizes the second set of ball drops. As one might expect, the whiffle balls all reach the ground later than the other balls. This is to be expected since the openings in their faces increase the drag they experience. Even so, all the balls reach the ground within a second of each other. We also plot the simulated trajectories of two spheres falling with constant linear (in v) drag and the trajectory predicted by constant acceleration. Note that, based on the log-log plot of displacement, none of the balls appears to have reached terminal velocity by the time they hit the ground. This may increase the difficulty of accurately inferring the balls' governing equations. Given only measurements from one regime of falling ball dynamics, it may prove difficult to infer models that generalize to other regimes.

Drawing inspiration from Aristotle, one might form the hypothesis that the amount of time taken by spheres to reach the ground should be a function of the *density* of the spheres. Density takes into account both information about the mass of an object and its volume,

Ball	**Radius**	**Mass**	**Density**	**Max vel.**	**Max Re**
Golf Ball	0.02196	0.0454	1022.07	26.63	1.75×10^5
Baseball	0.03541	0.1417	762.04	26.61	2.83×10^5
Tennis Ball	0.03303	0.0567	375.81	21.95	2.18×10^5
Volleyball	0.105*	NA	NA	22.09	6.96×10^5
Blue Basketball	0.11937	0.5103	71.63	24.80	8.88×10^5
Green Basketball	0.11658	0.4536	68.34	25.06	8.77×10^5
Whiffle Ball 1	0.03629	0.0283	141.64	16.91	1.84×10^5
Whiffle Ball 2	0.03629	0.0283	141.64	16.35	1.78×10^5
Yellow Whiffle Ball	0.04616	0.0425	103.25	15.30	2.12×10^5
Orange Whiffle Ball	0.04616	0.0425	103.25	15.77	2.18×10^5

Table 2.1: Physical measurements, maximum velocities across the two drops, and maximum Reynolds numbers for the dropped balls. Radius is measured in meters, mass in kilograms, density in kilograms per meter and maximum velocity in meters per second. *We do not have measurement data for the volleyball, but obtained an estimate for its radius based on other volleyballs in order to approximate its maximum Reynolds number.

which might be thought to affect the air resistance it encounters. We plot the landing time of each ball as a function of its density for both drops in Figure 2.4. To be more precise, because some balls were dropped from slightly different heights, we measure the amount of time it takes each ball to travel a fixed distance after being dropped, not the amount of time it takes the ball to reach the ground. There is a general trend across the tests for the denser balls to travel faster. However, the basketballs defy this trend and complete their journeys about as quickly as the densest ball. This shows there must be more factors at play than just density. There is also variability in the land time of the balls across drops. While most of the balls have very consistent fall times across drops, the blue basketball, golf ball, and orange whiffle ball reach the finish line faster in the first trial than the second one. These differences could be due to a variety of factors, including the balls being released with different initial

Figure 2.2: The balls that were dropped from the bridge, with the volleyball omitted. From left to right: Golf Ball, Tennis Ball, Whiffle Ball 1, Whiffle Ball 2, Baseball, Yellow Whiffle Ball, Orange Whiffle Ball, Green Basketball, and Blue Basketball. The two colored whiffle balls have circular openings and are structurally identical. The two white whiffle balls have elongated slits and are also identical.

velocities, or errors in measuring the balls' heights.

There are multiple known sources of error in the measurement data. The relatively low resolution of the videos means that the inferred ball heights are only approximate. In Section 2.5.5 we attempt to infer the level of noise introduced by our use of heights derived from imperfect video data. Furthermore, the camera was held by a person, not mounted on a tripod, leading to shaky footage. The true bridge height is uncertain because it was measured with a laser range finder claiming to be accurate to within 0.5 meters. Because the experiments were executed outside, it is possible for any given drop to have been affected by wind. Detecting exactly when each ball was dropped, at what velocity it was dropped, and when it hit the ground using only videos is certain to introduce further error. Finally, treating these balls as perfect spheres is an approximation whose accuracy depends on the nature of the balls. This idealization seems least appropriate for the whiffle balls, which are sure to exhibit much more complicated aerodynamic effects than, say, the baseball. The bowling ball was excluded from consideration because of corrupted measurements from its first drop.

The situation we strive to mimic with this experiment is one in which the researcher is in a position of ignorance about the system being studied. In order to design an experiment which eliminates the effects of confounding factors such as air resistance one must already

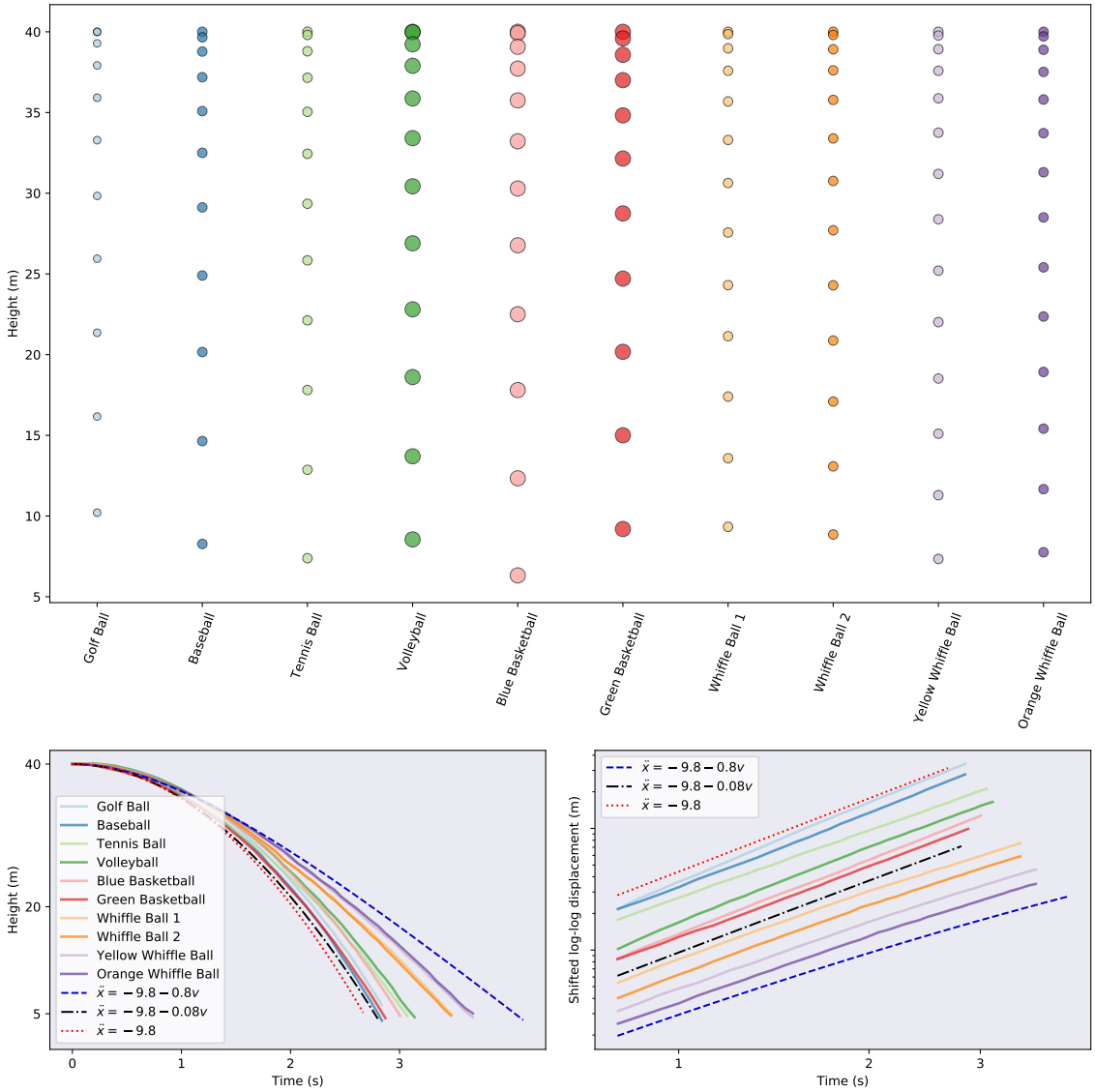


Figure 2.3: Visualizations of the ball trajectories for the second drop. Top: Subsampled raw drop data for each ball. Bottom left: Height for each ball as a function of time. We also include the simulated trajectories of idealized balls with differing levels of drag (black and blue) and a ball with constant acceleration (red). Bottom right: A log-log plot of the displacement of each ball from its original position atop the bridge. Note that we have shifted the curves vertically and zoomed in on the later segments of the time series to enable easier comparison. In this plot a ball falling at a constant rate (zero acceleration) will have a trajectory represented by a line with slope one. A ball falling with constant acceleration will have a trajectory represented by a line with slope two. A ball with drag will have a trajectory which begins with slope two and asymptotically approaches a line with slope one.

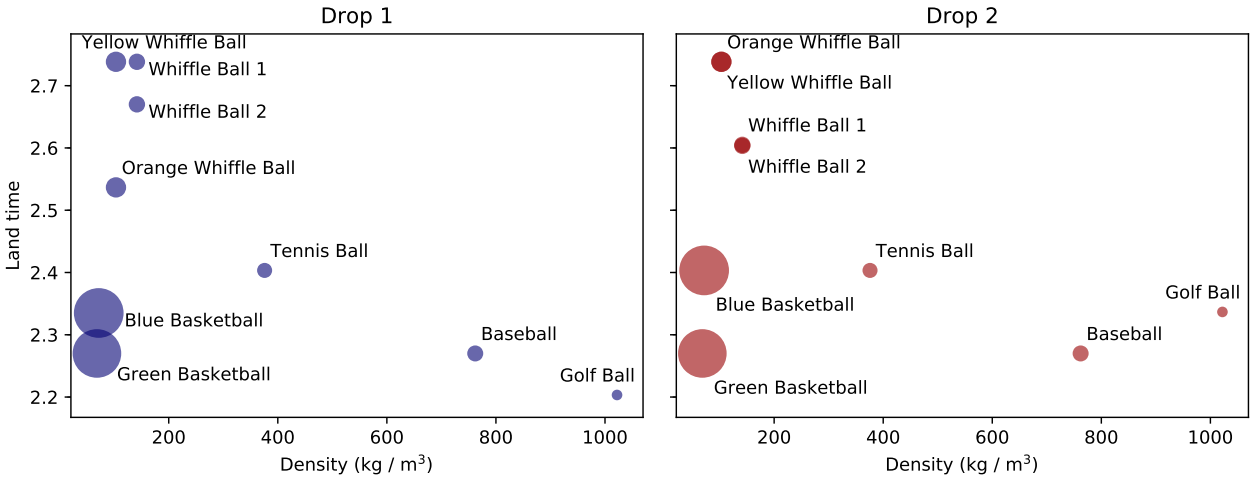


Figure 2.4: The amount of time taken by each ball to travel a fixed distance as a function of ball density.

have an appreciation for which factors are worth controlling; one leverages prior knowledge as Galileo did when he employed ramps in his study of falling objects to mitigate the effect of air resistance. In the early stages of investigation of a physical phenomenon, one must often perform poorly-controlled experiments to help identify these factors. We view the ball drop trials as this type of experiment.

In addition to the measurement data just described, we construct a synthetic data set by simulating falling objects with masses of 1 kg and different (linear) drag coefficients. In particular, for each digital ball, we simulate two drops of the same length as the real data and collect height measurements at a rate of 15 measurements per second. The balls fall according to the equation $\ddot{x}(t) = -9.8 + D\dot{x}(t)$, with each ball having its own *constant* drag coefficient, $D < 0$. We simulate five balls in total, with respective drag coefficients -0.1, -0.3, -0.3, -0.5, and -0.7. These coefficients are all within the plausible range suggested by the simulated trajectories shown in Figure 2.3. Each object is "dropped" with an initial velocity of 0. Varying amounts of Gaussian noise are added to the height data so that we may better explore the noise tolerance of the proposed model discovery approaches:

$$\tilde{x}_i = x_i + \eta\epsilon_i.$$

where $\eta \geq 0$ and $\epsilon_i \sim N(0,1)$; that is to say ϵ_i is normally distributed with unit variance.

2.3 Methods

In this section we describe the model discovery methods we employ to infer governing equations from noisy data. We first give the mathematical background necessary for learning dynamics via sparse regression and provide a brief overview of the SINDy method in Section 2.3.1. In Section 2.3.2 we propose a group sparsity regularization strategy for improving the robustness and generalizability of SINDy. We briefly discuss the setup of the model discovery problem we are attempting to solve in Section 2.3.3. Finally, we discuss numerical differentiation, a subroutine critical to effective model discovery, in Section 2.3.4.

2.3.1 Sparse identification of nonlinear dynamical systems

Consider the nonlinear dynamical system for the state vector $\mathbf{x}(t) = [x_1(t), x_2(t), \ldots, x_n(t)]^\top \in \mathbb{R}^n$ defined by

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

Given a set of noisy measurements of $\mathbf{x}(t)$, the sparse identification of nonlinear dynamics (SINDy) method, introduced in [21], seeks to identify $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. In this section we give an overview of the steps involved in the SINDy method and the assumptions upon which it relies. Throughout this chapter we refer to this algorithm as the *unregularized SINDy* method, not because it involves no regularization, but because its regularization is not as closely tailored to the problem at hand as the method proposed in Section 2.3.2.

For many dynamical systems of interest, the function specifying the dynamics, $\mathbf{f}$, consists of only a few terms. That is to say, when represented in the appropriate basis, there is a sense in which it is sparse. The key idea behind the SINDy method is that if one supplies a rich enough set of candidate functions for representing $\mathbf{f}$, then the correct terms can be identified using sparse regression techniques. The explicit steps are as follows. First we collect a set of (possibly noisy) measurements of the state $\mathbf{x}(t)$ and its derivative $\dot{\mathbf{x}}(t)$ at a sequence of

points in time, $t_1, t_2, \ldots, t_m$. These measurements are concatenated into two matrices, the columns of which correspond to different state variables and the rows of which correspond to points in time.

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}(t_1)^\top \\ \mathbf{x}(t_2)^\top \\ \vdots \\ \mathbf{x}(t_m)^\top \end{bmatrix} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \ldots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \ldots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \ldots & x_n(t_m) \end{bmatrix},$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}(t_1)^\top \\ \dot{\mathbf{x}}(t_2)^\top \\ \vdots \\ \dot{\mathbf{x}}(t_m)^\top \end{bmatrix} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \ldots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \ldots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \ldots & \dot{x}_n(t_m) \end{bmatrix}.$$

Next we specify a set of candidate functions, $\{\phi_i(\mathbf{x}) : i = 1, 2, \ldots, p\}$, with which to represent $\mathbf{f}$. Examples of candidate functions include monomials up to some finite degree, trigonometric functions, and rational functions. In practice the selection of these functions can be informed by the practitioner's prior knowledge about the system being measured. The candidate functions are evaluated on $\mathbf{X}$ to construct a library matrix

$$\mathbf{\Phi}(\mathbf{X}) = \begin{bmatrix} | & | & & | \\ \phi_1(\mathbf{X}) & \phi_2(\mathbf{X}) & \ldots & \phi_p(\mathbf{X}) \\ | & | & & | \end{bmatrix}.$$

Note that each column of $\mathbf{\Phi}(\mathbf{X})$ corresponds to a single candidate function. Here we have overloaded notation and interpret $\phi(\mathbf{X})$ as the column vector obtained by applying ϕ_i to each row of $\mathbf{X}$. It is assumed that each component of $\mathbf{f}$ can be represented as a *sparse* linear combination of such functions. This allows us to pose a regression problem to be solved for the coefficients used in these linear combinations:

$$\dot{\mathbf{X}} = \mathbf{\Phi}(\mathbf{X})\mathbf{\Xi}. \tag{2.2}$$

We adopt MATLAB-style notation and use $\mathbf{\Xi}_{(:,j)}$ to denote the j-th column of $\mathbf{\Xi}$. The

coefficients specifying the dynamical system obeyed by $\mathbf{x}_j$ are stored in $\boldsymbol{\Xi}_{(:,j)}$:

$$\dot{\mathbf{x}}_j = \mathbf{f}_j(\mathbf{x}) = \boldsymbol{\Phi}\left(\mathbf{x}^\top\right)\boldsymbol{\Xi}_{(:,j)},$$

where $\boldsymbol{\Phi}\left(\mathbf{x}^\top\right)$ is to be interpreted as a (row) vector of symbolic functions of components of $\mathbf{x}$. The full system of differential equations is then given by

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) = \boldsymbol{\Xi}^\top\left(\boldsymbol{\Phi}\left(\mathbf{x}^\top\right)\right)^\top.$$

For concreteness we supply the following example. With the candidate functions $\{1, x_1, x_2, x_1x_2, x_1^2, x_2^2\}$ the Lotka-Volterra equations

$$\begin{cases} \dot{x}_1 = & \alpha x_1 - \beta x_1 x_2, \\ \dot{x}_2 = & \delta x_1 x_2 - \gamma x_2 \end{cases}$$

can be expressed as

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \boldsymbol{\Xi}^\top\left(\boldsymbol{\Phi}\left(\mathbf{x}^\top\right)\right)^\top = \begin{bmatrix} 0 & \alpha & 0 & -\beta & 0 & 0 \\ 0 & 0 & -\gamma & \delta & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

Were we to obtain pristine samples of $\mathbf{x}(t)$ and $\dot{\mathbf{x}}(t)$ we could solve (2.2) exactly for $\boldsymbol{\Xi}$. Furthermore, assuming we chose linearly independent candidate functions and avoided collecting redundant measurements, $\boldsymbol{\Xi}$ would be unique and would exhibit the correct sparsity pattern. In practice, however, measurements are contaminated by noise and we actually observe a perturbed version of $\mathbf{x}(t)$. In many cases $\dot{\mathbf{x}}(t)$ is not observed directly and must instead be approximated from $\mathbf{x}(t)$, establishing another source of error. The previously exact equation, (2.2), to be solved for $\boldsymbol{\Xi}$ is supplanted by the approximation problem

$$\dot{\mathbf{X}} \approx \boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Xi}.$$

To find $\boldsymbol{\Xi}$ we solve the more concrete optimization problem

$$\min_{\boldsymbol{\Xi}} \frac{1}{2} \left\| \dot{\mathbf{X}} - \boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Xi} \right\|_F^2 + \Omega(\boldsymbol{\Xi}), \tag{2.3}$$

where $\Omega(\cdot)$ is a regularization term chosen to promote sparse solutions and $\|\cdot\|_F$ is the Frobenius norm. Note that because any given column of $\boldsymbol{\Xi}$ encodes a differential equation for a single component of $\mathbf{x}$, each column generates a problem that is decoupled from the problems associated with the other columns. Thus, solving (2.3) consists of solving n separate regularized least squares problems. Row i of $\boldsymbol{\Xi}$ contains the coefficients of library function ϕ_i for each governing equation.

The most direct way to enforce sparsity is to choose Ω to be the ℓ_0 penalty, defined as $\|\mathbf{M}\|_0 = \sum_{i,j} |\text{sign}(M_{ij})|$. This penalty simply counts the number of nonzero entries in a matrix or vector. However, using the ℓ_0 penalty makes (2.3) difficult to optimize because $\|\cdot\|_0$ is nonsmooth and nonconvex. Another common choice is the ℓ_1 penalty defined by $\|\mathbf{M}\|_1 = \sum_{i,j} |M_{ij}|$. This function is the convex relaxation of the ℓ_0 penalty. The LASSO, proposed in [119], with coordinate descent is typically employed to solve (2.3) with $\Omega(\cdot) = \|\cdot\|_1$, but this method can become computationally expensive for large data sets and often leads to incorrect sparsity patterns [111]. Hence we solve (2.3) using the sequential thresholded least-squares algorithm proposed in [21], and studied in further detail in [134]. In essence, the algorithm alternates between (a) successively solving the *unregularized* least-squares problem for each column of $\boldsymbol{\Xi}$ and (b) removing candidate functions from consideration whose corresponding components in $\boldsymbol{\Xi}$ are below some threshold. This threshold or sparsity parameter, is straightforward to interpret: no governing equations are allowed to have any terms with coefficients of magnitude smaller than the threshold. Crucially, it should be noted that just because a candidate function is discarded for one column of $\boldsymbol{\Xi}$ (i.e. for one component's governing equation) does not mean it is removed from contention for the other columns. A simple Python implementation of sequentially thresholded least-squares is provided in Section 2.5.1.

We note that if we simulate falling objects with constant acceleration, $\ddot{x}(t) = -9.8$, or

linear drag, $\ddot{x}(t) = -9.8 + D\dot{x}(t)$, and add *no noise*, then there is almost perfect agreement between the true governing equations and the models learned by SINDy. Section 2.5 contains a more thorough discussion of such numerical experiments and another example application of SINDy.

SINDy has a number of well-known limitations. The biggest of these is the effect of noise on the learned equations. If one does not have direct measurements of derivatives of state variables, then these derivatives must be computed numerically. Any noise that is present in the measurement data is amplified when it is numerically differentiated, leading to noise in both $\dot{\mathbf{X}}$ and $\mathbf{\Phi}(\mathbf{X})$ in (2.3). In its original formulation, SINDy often exhibits erratic performance in the face of such noise, but extensions have been developed which handle noise more gracefully [101, 120]. We discuss numerical differentiation further in Section 2.3.4. As with other methods, each degree of freedom supplied to the practitioner presents a potential source of difficulty. To use SINDy one must select a set of candidate functions, a sparse regularization function, and a parameter weighing the relative importance of the sparseness of the solution against accuracy. An improper choice of any one of these can lead to poor performance. The set of possible candidate functions is infinite, but SINDy requires one to specify a finite number of them. If one has any prior knowledge of the dynamics of the system being modeled, it can be leveraged here. If not, it is typically recommended to choose a class of functions general enough to encapsulate a wide variety of behaviors (e.g. polynomials or trigonometric functions). In theory, sparse regression techniques should allow one to specify a sizable library of functions, selecting only the relevant ones. However, in practice, the underlying regression problem becomes increasingly ill-conditioned as more functions are added. If one wishes to explore an especially large space of possible library functions it may be better to use other approaches such as symbolic regression with genetic algorithms [13, 104]. A full discussion of how to pick a sparsity-promoting regularizer is beyond the scope of this work. We do note that there have been recent efforts to explore different methods for obtaining sparse solutions when using SINDy [28]. An appropriate value for the sparsity hyperparameter can be obtained using cross-validation. We note that the need to perform

hyperparameter tuning is by no means unique to SINDy. Virtually all machine learning methods require some amount of hyperparameter tuning. There are two natural options for target metrics during cross-validation. The derivatives directly predicted by the linear model can be compared against the measured (or numerically computed) derivatives. Alternatively, the model can be fed into a numerical integrator along with initial conditions to obtain predicted future values for the state variables. These forecasts can then be judged against the measured values. To achieve a balance between model sparsity and accuracy, information theoretic criteria such as the Akaike information criteria (AIC) or Bayes information criteria (BIC) can be applied [68].

2.3.2 Group sparsity regularization

The standard, unregularized SINDy approach attempts to learn the dynamics governing each state variable independently. It does not take into account prior information one may possess regarding relationships between state variables. Intuitively speaking, the balls in our data set (whiffle balls, perhaps, excluded) are similar enough objects that the equations governing their trajectories should include similar terms. In this subsection we propose a group sparsity method which can be interpreted as enforcing this hypothesis when seeking predictive models for the balls.

We draw inspiration for our approach from the group LASSO of [133], which extends the LASSO. The classic LASSO method solves the ℓ_1 regularization problem

$$\beta = \arg\min_{\beta} \frac{1}{2} \|\mathbf{X}\beta - \mathbf{Y}\|_2^2 + \lambda \|\beta\|_1 . \tag{2.4}$$

which penalizes the magnitude of each component of β *individually*. The group LASSO approach modifies (2.4) by bundling sets of related entries of β together when computing the penalty term. Let the entries of β be partitioned into G disjoint blocks $\{\beta_1, \beta_2, \ldots, \beta_G\}$, which can be treated as vectors. The group LASSO then solves the following optimization problem

$$\beta = \arg\min_{\beta} \frac{1}{2} \|\mathbf{X}\beta - \mathbf{Y}\|_2^2 + \lambda \sum_{i=1}^{G} \|\beta_i\|_2 . \tag{2.5}$$

In the case that the groups each consist of exactly one entry of β, (2.5) reduces to (2.4). When blocks contain multiple entries, the group LASSO penalty encourages them to be retained or eliminated as a group. Furthermore, it drives sets of unimportant variables to truly vanish, unlike the ℓ_2 regularization function which merely assigns small but nonzero values to insignificant variables.

We apply similar ideas in our *group sparsity* method for the SINDy framework and force the models learned for each ball to select the same library functions. Recall that the model variables are contained in $\boldsymbol{\Xi}$. To enforce the condition that each governing equation should involve the same terms, we identify *rows* of $\boldsymbol{\Xi}$ as sets of variables to be grouped together. Borrowing MATLAB notation again, we let $\boldsymbol{\Xi}_{(i,:)}$ denote row i of $\boldsymbol{\Xi}$. To perform sequential thresholded least squares with the group sparsity constraint we repeatedly apply the following steps until convergence: (a) solve the least-squares problem (2.3) *without* a regularization term for each column of $\boldsymbol{\Xi}$ (i.e. for each ball), (b) prune the library, $\boldsymbol{\Phi}(\mathbf{X})$, of functions which have low relevance across most or all of the balls. This procedure is summarized in Algorithm 1.

Algorithm 1: A group sparsity algorithm for the sequential thresholded least squares method

Data: $\dot{\mathbf{X}} \in \mathbb{R}^{m\times d}$, $\mathbf{\Phi}(\mathbf{X}) \in \mathbb{R}^{m\times p}$, and $\delta > 0$

Result: coefficient matrix $\mathbf{\Xi} \in \mathbb{R}^{p\times d}$

while *not converged* **do**

  `// Solve a least squares problem for each state variable`

  for $j \leftarrow 1$ **to** d **do**

    $\mathbf{\Xi}_{(:,j)} \leftarrow \arg\min_{\xi} \frac{1}{2} \left\| \dot{\mathbf{X}} - \mathbf{\Phi}(\mathbf{X})\xi \right\|_2^2$;

  end

  `// Remove library functions with low salience`

  for $i \leftarrow 1$ **to** p **do**

    if $R(\mathbf{\Xi}_{(i,:)}) < \delta$ **then**

      Delete $\mathbf{\Xi}_{(i,:)}$ and $\mathbf{\Phi}(\mathbf{X})_{(:,i)}$;

    end

  end

end

Replace deleted rows of $\mathbf{\Xi}$ and deleted columns of $\mathbf{\Phi}(\mathbf{X})$ with 0's;

Here R is a function measuring the importance of a row of coefficients. Possible choices for R include the ℓ_1 or ℓ_2 norm of the input, the mean or median of the absolute values of the entries of the input, or another statistical property of the input entries such as the lower 25% quantile. In this work we use the ℓ_1 norm. Convergence is attained when no rows of $\mathbf{\Xi}$ are removed. Note that while all the models are constrained to be generated by the same library functions, the *coefficients* in front of each can differ from one model to the next. The hyperparameter δ controls the sparsity of $\mathbf{\Xi}$, though not as directly as the sparsity parameter for SINDy. Increasing it will result in models with fewer terms and decreasing it will have the opposite effect. Since we use the ℓ_1 norm and there are 10 balls in our primary data set, rows of $\mathbf{\Xi}$ whose average magnitude is less than $\frac{\delta}{10}$ are removed.

Because the time series are all noisy, it is likely that some the differential equations returned by the unregularized SINDy algorithm will acquire spurious terms. Insisting that only terms which *most* of the models find useful are kept, as with our group sparsity method, should help to mitigate this issue. In this way we are able to leverage the fact that we have multiple trials involving similar objects to improve the robustness of the learned models to noise. Even if some of the unregularized models from a given drop involve erroneous library functions, we might still hope that, on average, the models will pick the correct terms. Our approach can also be viewed as a type of *ensemble* method wherein a set of models is formed from the time series of a given drop, they are allowed to vote on which terms are important, then the models are retrained using the constrained set of library functions agreed upon in the previous step.

2.3.3 Equations of motion

Even the simplest model for the height, $x(t)$, of a falling object involves an acceleration term. Consequently, we impose the restriction that our model be a second order (autonomous) differential equation:

$$\ddot{x} = f(x, \dot{x}). \tag{2.6}$$

The SINDy framework is designed to work with first order systems of differential equations, so we convert (2.6) into such a system:

$$\begin{cases} \dot{x} = & v \\ \dot{v} = & g(x, v). \end{cases}$$

We then apply SINDy, with $\mathbf{x} = \begin{bmatrix} x & v \end{bmatrix}^\top$ and $f(\mathbf{x}) = \begin{bmatrix} v & g(\mathbf{x}) \end{bmatrix}^\top$, and attempt to learn the function g. In fact, because we already know the correct right-hand side function for $\dot{x}$, we need only concern ourselves with finding an expression for $\dot{v}$.

Our nonlinear library consists of polynomials in x and v up to degree three, visualized in

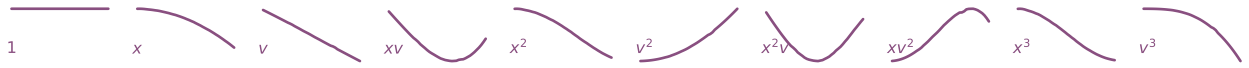


Figure 2.5: Visualizations of nonlinear library functions corresponding to the second green basketball drop. If the motion of the balls is described by Newton's second law, $F = m\ddot{x}$, then these functions can be interpreted as possible forcing terms constituting F.

Figure 2.5:

$$\mathbf{\Phi}(\mathbf{X}) = \begin{bmatrix} | & | & | & | & | & | & & | \\ \mathbf{1} & \mathbf{x}(t) & \mathbf{v}(t) & \mathbf{x}(t)\mathbf{v}(t) & \mathbf{x}(t)^2 & \mathbf{v}(t)^2 & \dots & \mathbf{v}(t)^3 \\ | & | & | & | & | & | & & | \end{bmatrix}. \tag{2.7}$$

Assuming that the motion of the balls is completely determined by Newton's second law, $F = ma = m\ddot{x}$, we may interpret the SINDy algorithm as trying to discover the force (after dividing by mass) that explains the observed acceleration.

Though we know now that the acceleration of a ball should not depend on its height, we seek to place ourselves in a position of ignorance analogous to the position scientists would have found themselves in centuries ago. We leave it to our algorithm to sort out which terms are appropriate. In practice one might selectively choose which functions to include in the library based on domain knowledge, or known properties of the system being modeled.

2.3.4 Numerical differentiation

In order to form the nonlinear library (2.7) and the derivative matrix, $\dot{\mathbf{X}}$, we must approximate the first two derivatives of the height data from each drop. Applying standard numerical differentiation techniques to a signal amplifies any noise that is present. This poses a serious problem since we aim to fit a model to the *second* derivative of the height measurements. Because the amount of noise in our data set is nontrivial, two iterations of numerical differentiation will create an intolerable noise level. To mitigate this issue we apply a Savitzky-Golay filter from [99] to smooth the data before differentiating via second order

centered finite differences. Points in a noisy data set are replaced by points lying on low-degree polynomials which are fit to localized patches of the original data with a least-squares method. Other available approaches include using a total variation regularized derivative as in [21] or working with an integral formulation of the governing equations as described in [101]. We perform a detailed analysis of the error introduced by smoothing and numerical differentiation in Section 2.5.4.

2.4 Results

2.4.1 Learned terms

In this section we compare the terms present in the governing equations identified using the unregularized SINDy approach with those present when the group sparsity constraint is imposed. We train separate models on the two drops. The two algorithms are given one sparsity hyperparameter each to be applied for all balls in both drops. The group sparsity method used a value of 1.5 and the other method used a value of 0.04. These parameters were chosen by hand to balance allowing the algorithms enough expressiveness to model the data, while being restrictive enough to prevent widespread overfitting; increasing them produces models with one or no terms and decreasing them results in models with large numbers of terms. See Section 2.5.6 for a more detailed discussion of our choice of sparsity parameter values.

Figure 2.6 summarizes the results of this experiment. Learning a separate model for each ball independent of the others allows many models to fall prey to overfitting. Note how most of the governing equations incorporate an extraneous height term. On the other hand, two of the learned models involve only constant acceleration and fail to identify any effect resembling air resistance.

The method leveraging group sparsity is more effective at eliminating extraneous terms and selecting only those which are useful across most balls. Moreover, only the constant and velocity terms are active, matching our intuition that the dominant forces at work are

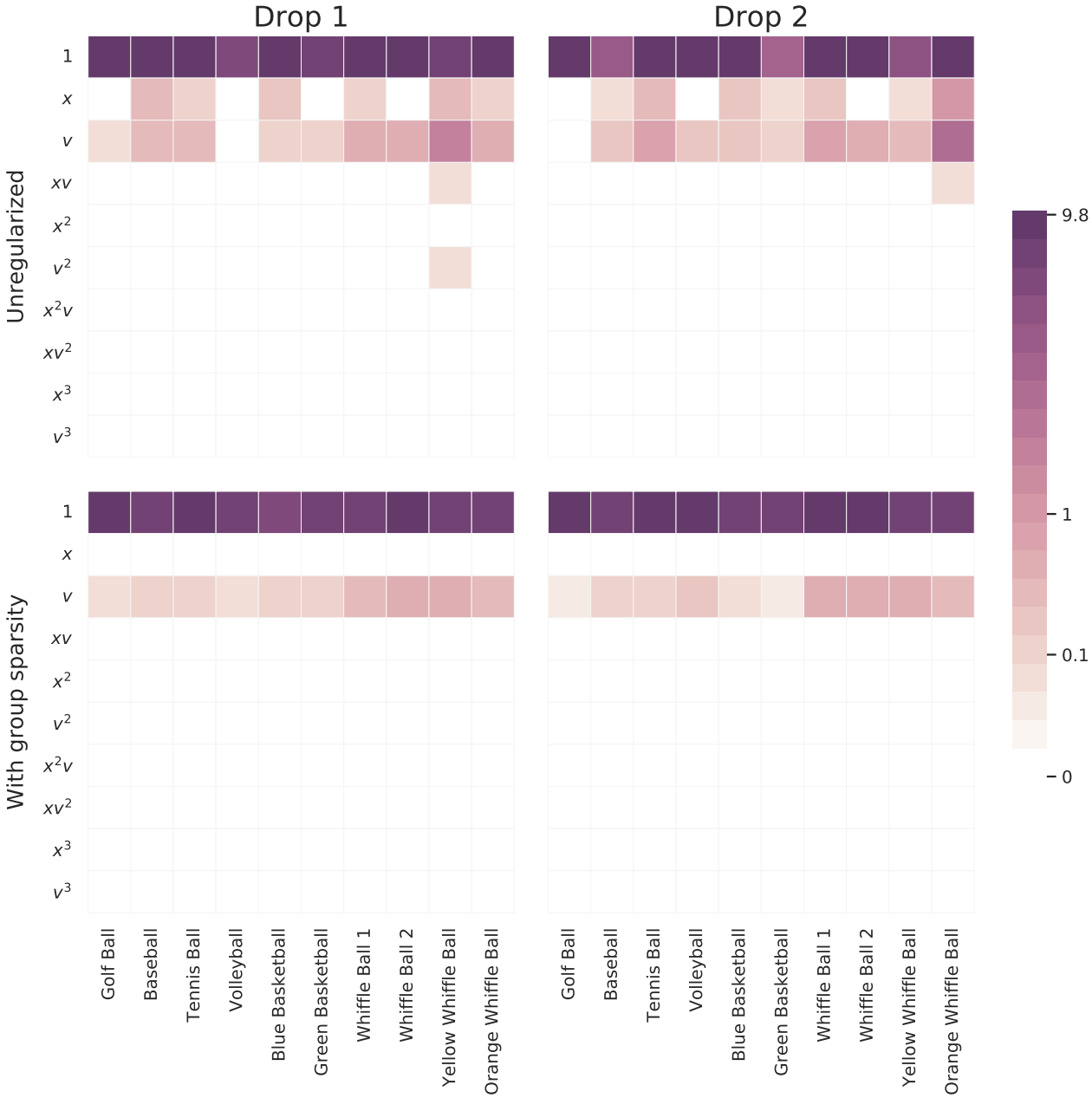


Figure 2.6: Magnitudes of the coefficients learned for each ball by models trained on one drop either with or without the proposed group sparsity approach. The unregularized approach used a sparsity parameter of 0.04 and the group sparsity method used a value of 1.5. Increasing this parameter slightly in the unregularized case serves to push many models to use only a constant function.

gravity and drag due to air resistance. Interestingly, the method prefers a linear drag term, one proportional to v, to model the discrepancy between measured trajectories and constant acceleration. Even the balls which don't include a velocity term in the unregularized model have this term when group sparsity regularization is employed. This shows that group penalty can simultaneously help to dismiss distracting candidate functions and promote correct terms that may have been overlooked. Is is also reassuring to see that, compared to the other balls, the whiffle ball models have larger coefficients on the v terms. Their accelerations slow at a faster rate as a function of their velocities than do the other balls.

The actual governing equations learned with the group sparsity method are provided in Table 2.2. Every equation has a constant acceleration term within a few meters per second squared of -9.8, but few are quite as close as one might expect. Thus even with a stable method of inferring governing equations, based on this data one would not necessarily conclude that all balls experience the same (mass-divided) force due to gravity. Note also that some of the balls mistakenly adopt *positive* coefficients multiplying v. The balls for which this occurs tend to be those whose motion is well-approximated by constant acceleration. Because the size of the discrepancy between a constant acceleration model and these balls' measured trajectories is not much larger than the amount of error suspected to be present in the data, SINDy has a difficult time choosing an appropriate value for the v terms. One would likely need higher resolution, higher accuracy measurement data in order to obtain reasonable approximations of the drag coefficients or v^2 terms.

At 65 degrees Fahrenheit, the density of air ρ at sea level is 1.211kg/m^3 [128] and its dynamic viscosity μ is $1.82 \times 10^{-5}\text{kg/(m s)}$. The Reynolds number for a ball with diameter D and velocity v will then be

$$Re = 0.667Dv \times 10^5.$$

Table 2.1 gives the maximum velocities of each ball over the two drops and the corresponding Reynolds numbers. Note that these are the *maximum* Reynolds numbers, not the Reynolds numbers over the entire trajectories. With velocities under 30m/s and diameters from 0.04m to 0.22m we should expect Reynolds numbers with magnitudes ranging from 10^4 to 10^5 over

Ball	**First drop**	**Second drop**
Golf Ball	$\ddot{x} = -9.34 + 0.05v$	$\ddot{x} = -9.44 - 0.03v$
Baseball	$\ddot{x} = -8.51 + 0.14v$	$\ddot{x} = -7.56 + 0.14v$
Tennis Ball	$\ddot{x} = -9.08 - 0.13v$	$\ddot{x} = -8.64 - 0.12v$
Volleyball	$\ddot{x} = -8.11 - 0.08v$	$\ddot{x} = -9.64 - 0.23v$
Blue Basketball	$\ddot{x} = -6.71 + 0.15v$	$\ddot{x} = -7.50 + 0.07v$
Green Basketball	$\ddot{x} = -7.36 + 0.10v$	$\ddot{x} = -8.05 + 0.02v$
Whiffle Ball 1	$\ddot{x} = -8.24 - 0.34v$	$\ddot{x} = -9.44 - 0.43v$
Whiffle Ball 2	$\ddot{x} = -9.81 - 0.56v$	$\ddot{x} = -9.79 - 0.48v$
Yellow Whiffle Ball	$\ddot{x} = -8.50 - 0.47v$	$\ddot{x} = -8.45 - 0.46v$
Orange Whiffle Ball	$\ddot{x} = -7.83 - 0.35v$	$\ddot{x} = -8.03 - 0.42v$

Table 2.2: Models learned by applying SINDy with group sparsity regularization (sparsity parameter $\delta = 1.5$) to each of the two ball drops.

the course of the balls' trajectories (apart from the very beginnings of each drop). The *average* trajectory consists of about 49 measurements, just over one of which corresponds to a Reynolds number that is $\mathcal{O}(10^3)$. About 13 of these measurements are associated with Reynolds numbers on the order of 10^4 and roughly 33 with Reynolds numbers of magnitude 10^5. Note that this means the majority of data points were collected when the balls were in the quadratic drag regime. Based on Figure 2.1 we should expect balls with Reynolds numbers less than 10^5 to have drag coefficients of magnitude about 0.5. Figure 2.1 suggests that balls experiencing higher Reynolds numbers such as the volleyball and basketballs should have smaller drag coefficients varying between 0.05 and 0.3 depending on their smoothness. The predicted (linear) drag coefficients for the volleyball lie in this range while the basketballs' learned drag coefficients are erroneously positive. If the basketballs are treated as being smooth, their drag coefficients predicted by Figure 2.1 may be too small for SINDy to identify given the noisy measurement data. A similar effect seems to occur for the golf ball.

Though it experiences a lower Reynolds number, its dimples induce a turbulent flow over its surface, granting it a small drag coefficient at a lower Reynolds number. Overall, the linear drag coefficients predicted by the model are at least within a physically reasonable range, with some outliers having incorrect signs.

Next we turn to the simulated data set. We perform the same experiment as with the real world data: we apply both versions of SINDy to a series of simulated ball drops and then note the models that are inferred. Our findings are shown in Figure 2.7. We need not say much about the standard approach: it does a poor job of identifying coherent models for all levels of noise. The group sparsity regularization is much more robust to noise, identifying the correct terms and their magnitudes for noise levels up to half a meter (in standard deviation). For more significant amounts of noise, even this method is unable to decide between adopting x or v into its models. Perhaps surprisingly, if a v^2 term with coefficient ~ 0.1 is added to the simulated model[1], the learned coefficients look nearly identical. Although this additional term visibly alters the trajectory (before it is corrupted by noise), none of the learned equations capture it, even in the absence of noise. One reason for this is because the coefficient multiplying v^2 is too small to be retained during the sequential thresholding least squares procedure. If we decrease the sparsity parameter enough to accommodate it, the models also acquire spurious higher order terms. To infer the v^2 term using the approach outlined here, one would need to design and carry out additional experiments which better isolate this effect, perhaps by using a denser fluid or by dropping a ball with a larger diameter of relatively small mass, thereby increasing the constant multiplying $v^2 C_D$ in (2.1). A much more realistic drag force based on (2.1) can be used to simulate falling balls. Such a drag force will shift from being linear to quadratic in v over the course of a ball's trajectory. In this scenario neither version of SINDy identifies a v^2 term, regardless of how much many measurements are collected, but both detect linear drag, exhibiting similar performance as

[1]It should be noted that, based on the balls' approximated velocities, the largest coefficient multiplying v^2 (i.e. $\frac{1}{2m}\rho A C_D$ from (2.1), where m is the mass of a ball), is less than 0.08 in magnitude, across all the trials.

is shown here. A more detailed discussion can be found in Section 2.5.7.

2.4.2 Model error

We now turn to the problem of testing the predictive performance of models learned from the data. We benchmark four models of increasing complexity on the drop data. The model templates are as follows:

1. Constant acceleration: $\ddot{x} = \alpha$
2. Constant acceleration with linear drag: $\ddot{x} = \alpha + \beta v$
3. Constant acceleration with linear and quadratic drag: $\ddot{x} = \alpha + \beta v + \gamma v^2$
4. Overfit model: Set a low sparsity threshold and allow SINDy to fit a more complicated model to the data

The model parameters α, β, and γ are learned using the SINDy algorithm using libraries consisting of just the terms required by the templates. The testing procedure consists of constructing a total of 80 models (4 templates $\times$ 10 balls $\times$ 2 drops) and then using them to predict a quantity of interest. First a template model is selected then it is trained using one ball's trajectory from one drop. Once trained, the model is given the initial conditions (initial height and velocity) from the same ball's other drop and tasked with predicting the ball's height after 2.8 seconds have passed[2]. Recall from Figure 2.4 that the same ball dropped twice from the same height by the same person on the same day can hit the ground at substantially different times. In the absence of any confounding factors, the time it takes a sphere to reach the ground after being released will vary significantly based on its initial velocity. Since there is sure to be some error in estimating the initial height and velocity of the balls, we should expect only modest accuracy in predicting their landing times. We summarize the outcome

[2]This number corresponds to the shortest set of measurement data across all the trials. All models are evaluated at 2.8 seconds to allow for meaningful comparison of error rates between models.

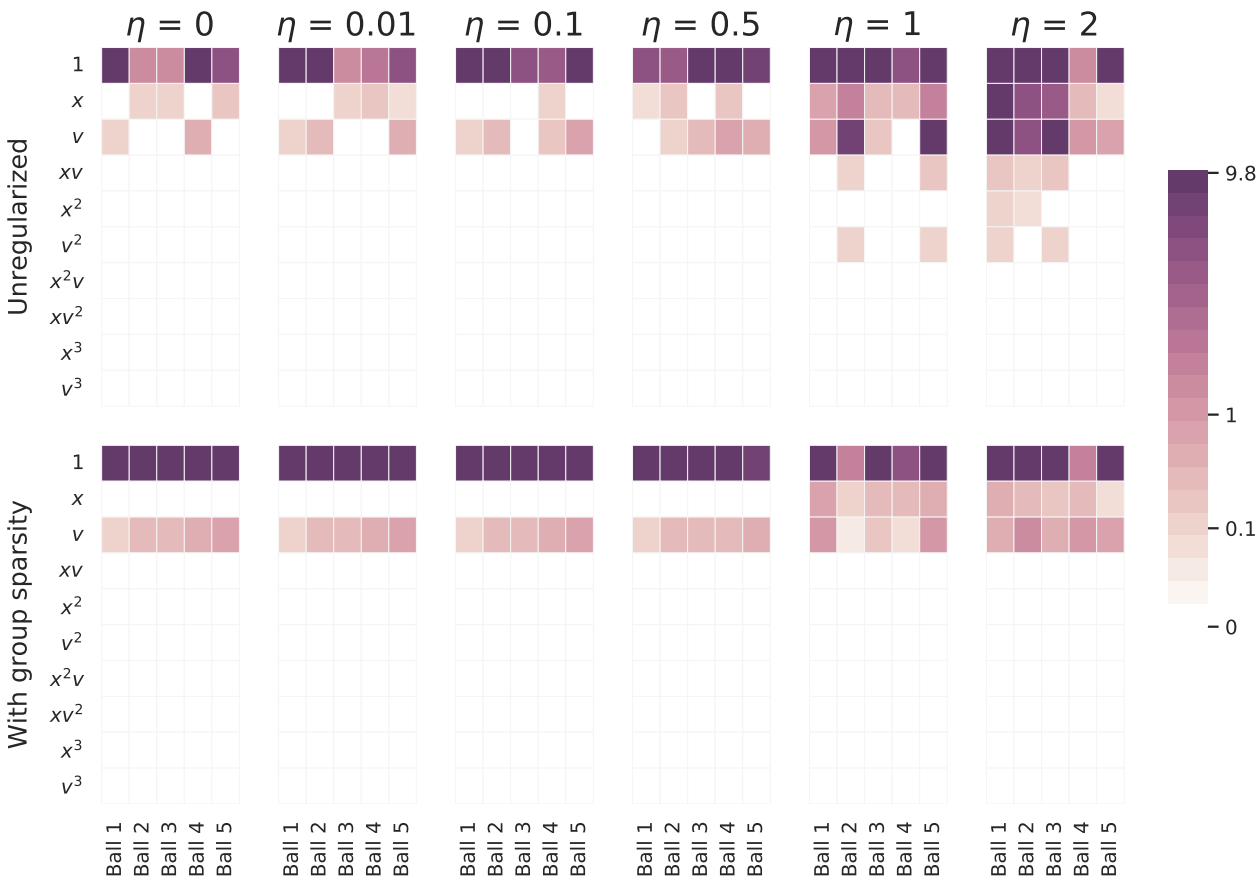


Figure 2.7: A comparison of coefficients of the models inferred from the simulated falling balls. The top row shows the coefficients learned with the standard SINDy algorithm and the bottom row the coefficients learned with the group sparsity method. η indicates the amount of noise added to the simulated ball drops. The standard approach used a sparsity parameter of 0.05 and the group sparsity method used a value of 1.5. The balls were simulated using constant acceleration and the following respective coefficients multiplying v: -0.1, -0.3, -0.3, -0.5, -0.7.

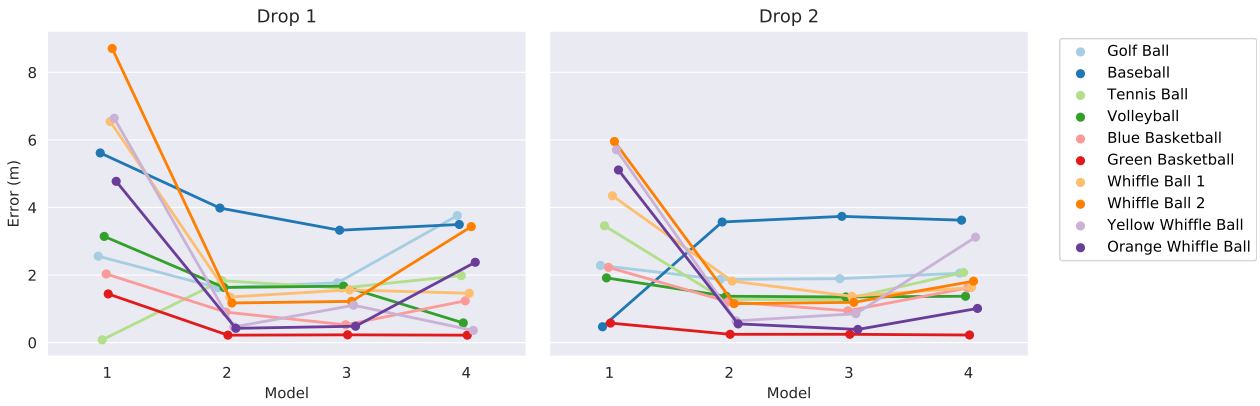


Figure 2.8: The error in landing time predictions for the four models. The results for the models trained on drops one and two are shown on the left and right, respectively. We have intentionally jittered the horizontal positions of the data points to facilitate easier comparison.

of this experiment in Figure 2.8. The error tends to decrease significantly between model one and model two, marking a large step in explaining the discrepancy between a constant acceleration model and observation. There does not appear to be a large difference between the predictive powers of models two and three as both seem to provide similar levels of accuracy. Occam's razor might be invoked here to motivate a preference for model two over model three since it is simpler and has the same accuracy. This provides further evidence that the level of noise and error in the data set is too large to allow one to accurately infer the dynamics due to v^2. Adding additional terms to the equations seems to weaken their generalizability somewhat, as indicated by the slight increase in errors for model four.

Figure 2.9 visualizes the forecasts of the learned equations for two of the balls along with their deviation from the true measurements. The models are first trained on data from drop 2, then they are given initial conditions from the same drop and made to predict the full trajectories. There are a few observations to be made. The constant acceleration models (model one) are clearly inadequate, especially for the whiffle ball. Their error is much higher than that of the other models indicating that they are underfitting the data, though constant acceleration appears to be a reasonable approximation for a falling golf ball. Models two

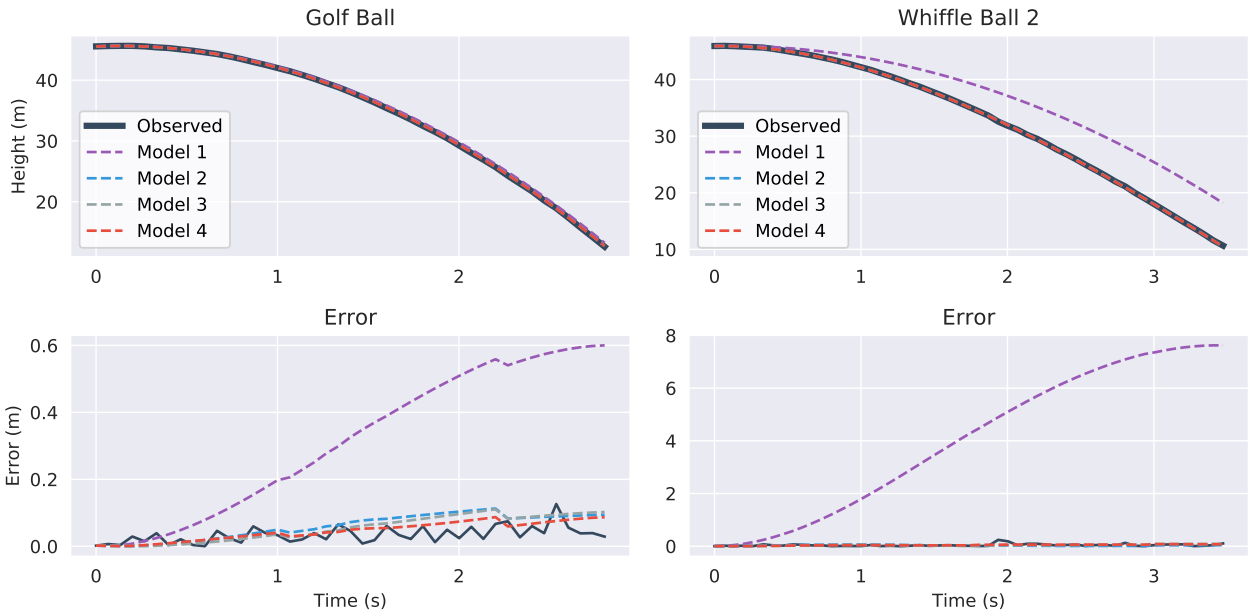


Figure 2.9: Predicted trajectories and error for the Golf Ball (top) and Whiffle Ball 2 (bottom). On the left we compare the predicted trajectories against the true path and on the right we show the absolute error for the predictions. The 'Observed' lines in the error plots show the difference between the original height measurements and the smoothed versions used for differentiation. They give an idea of the amount of intrinsic measurement noise. All models plotted were trained and evaluated on drop 2.

through four all seem to be imitating the trajectories to about the level of the measurement noise, which is about the most we could hope of them. It is difficult to say which model is best by looking at these plots alone. To break the tie we can observe what happens if we evaluate the models in "unfamiliar" circumstances and force them to extrapolate.

Supplying the same initial conditions as before, with initial height shifted up to avoid negative heights, we task the models with predicting the trajectories out to 15 seconds. The results are shown in Figure 2.10. All four models fit the observed data itself fairly well. However, six or seven seconds after the balls are released, a significant degree of separation has started to emerge between the trajectories. The divergence of the model four instances is the most abrupt and the most pronounced. The golf ball's model grows without bound after seven seconds. It is here that the danger of overfit, high-order models becomes obvious.

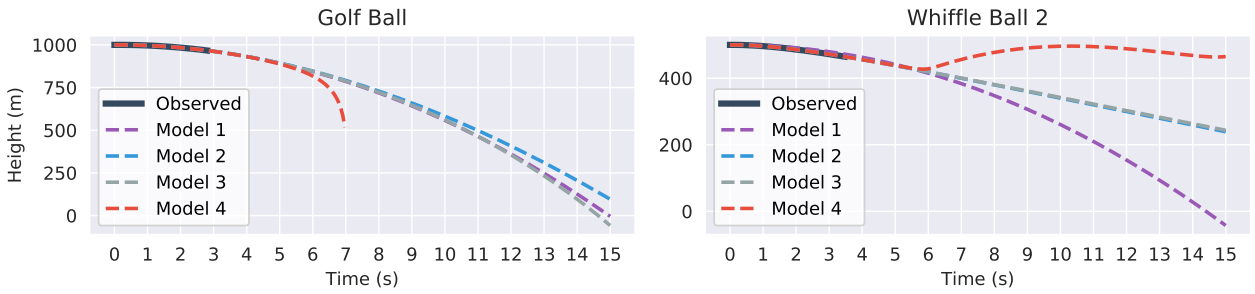


Figure 2.10: 15 second forecasted trajectories for the Golf Ball (left) and Whiffle Ball 2 (right) based on the second drop. Part of the graph of Model 4 (red) is omitted in the Golf Ball plot because it diverged to $-\infty$.

In contrast, the other models are better behaved. For the golf ball models one through three agree relatively well, perhaps showing that it is easier to predict the path of a falling golf ball than a falling whiffle ball. That model two is so similar to the constant acceleration of model one also suggests that the golf ball experiences very little drag. The v^2 term for model three has a coefficient which is erroneously positive and essentially cancels out the speed dampening effects of the drag term, leading to an overly rapid predicted descent. Models two and three agree extremely well for the whiffle ball as the learned v^2 coefficient is very small in magnitude.

2.5 Supplementary experiments

2.5.1 Sparse Identification of Nonlinear Dynamical systems: further details

In this section we provide some additional information concerning the Sparse Identification of Nonlinear Dynamical systems (SINDy) method. We first give a simplified implementation of the sequentially thresholded least-squares algorithm, implemented in Python, before showing examples of SINDy applied to two test problems: a nonlinear oscillator (Section 2.5.2) and a simulated falling body with different types of drag (Section 2.5.3).

Recall that to construct a set of governing equations, SINDy seeks to solve the following

optimization problem

$$\min_{\boldsymbol{\Xi}} \frac{1}{2} \left\| \dot{\mathbf{X}} - \boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Xi} \right\|_F^2 + \Omega(\boldsymbol{\Xi}), \tag{2.8}$$

where $\mathbf{X}$ is a matrix of measurements, $\dot{\mathbf{X}}$ is a matrix of derivatives of $\mathbf{X}$, $\boldsymbol{\Phi}(\mathbf{X})$ is a library matrix whose columns consist of potential right-hand side functions evaluated on the measurement data, $\boldsymbol{\Xi}$ is a coefficient matrix, and $\Omega(\cdot)$ is a regularization term encouraging sparsity. A one-dimensional version of the sequentially thresholded least-squares algorithm, which we use to solve (2.8) in this work[3], can be implemented in Python as

```
        xi = least_squares(theta, x_dot)  # Initial guess

        # delta is our sparsity parameter
        for k in range(iterations):
            small_indices = abs(xi) < delta
            big_indices = ~small_indices

            xi[small_indices] = 0  # Threshold small coefficients
            xi[big_indices] = least_squares(x[:, big_indices], x_dot)

```

Here we use `least_squares` to denote a black-box least-squares solver. This implementation only solves for one column of the coefficients of $\boldsymbol{\Xi}$, yielding the governing equation for only one measurement variable. In practice one runs this routine for each variable.

2.5.2 A brief example

Here we give an example of a dynamical system SINDy is easily able to identify: a first order nonlinear oscillator. The system is described by

$$\begin{aligned} \dot{x} &= -\frac{1}{10}x^3 + 2y^3 \\ \dot{y} &= -2x^3 - \frac{1}{10}y^3. \end{aligned} \tag{2.9}$$

[3]In actuality we use a custom implementation of sequentially thresholded least-squares for the majority of the results in this chapter and a recently developed package, PySINDy (`https://github.com/dynamicslab/pysindy`), for the examples in this section.

To construct training data for a SINDy model we simulate a trajectory under these dynamics starting from $(2, 0)$ for $t \in [0, 5]$ with a time step of 0.01. Using a threshold of 0.05 and a library consisting of polynomials terms of degree up to five, SINDy recovers the following model

$$\dot{x} = -0.100x^3 + 1.999y^3$$
$$\dot{y} = -1.999x^3 - 0.100y^3.$$

We plot the trajectories simulated from the actual model and the SINDy model for $t \in [0, 25]$ in Figure 2.11. Note the close agreement between the two trajectories.

2.5.3 Learning equations of motion

In this section we demonstrate that SINDy can readily learn simplified versions of the equations of motion, but struggles to identify dynamics containing terms not representable as linear combinations of the library terms. We simulate a ball of unit mass falling with constant acceleration and no drag

$$\dot{v} = -9.8, \qquad v(0) = 0$$

and with constant acceleration and linear drag

$$\dot{v} = -9.8 - 0.5v \qquad v(0) = 0.$$

Each simulation consists of 50 height measurements taken every fifteenth of a second. We numerically differentiate the height data, then feed the velocity profiles to SINDy models with thresholds of 0.1. SINDy learns the following governing equations:

$$\dot{v} = -9.8000 \qquad \text{(drag-free simulation)},$$
$$\dot{v} = -9.786 - 0.499v \qquad \text{(linear drag simulation)}.$$

Multiple factors contribute to the accuracy of the learned models for these two test cases. The effects of the constant acceleration and drag on the ball trajectories are relatively large,

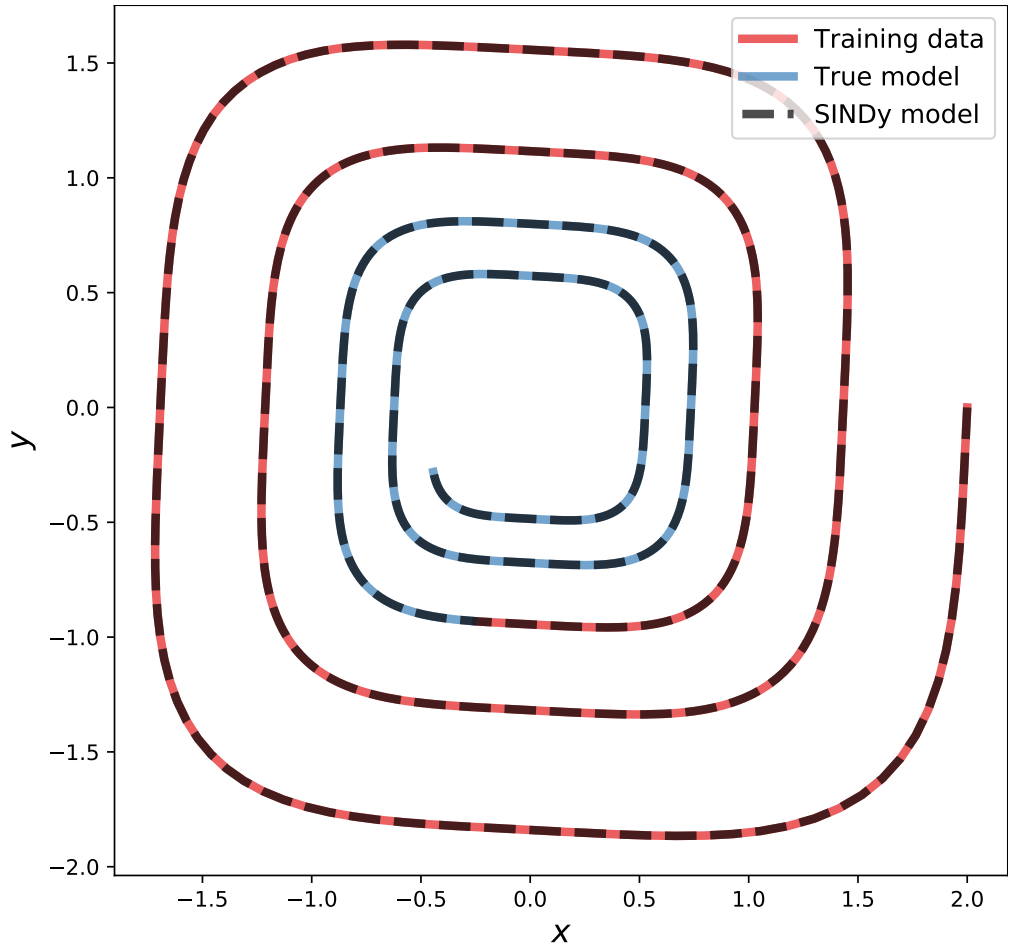


Figure 2.11: Dynamics of the nonlinear oscillator described by (2.9). The true trajectory, computed using (2.9) is plotted as a solid line, with red denoting the training data fed to the SINDy model and blue denoting the portion of the trajectory unseen by SINDy. The dashed line shows the dynamics predicted by the model discovered by the SINDy model starting at initial condition $(2, 0)$.

the data lacks noise, and the appropriate terms are present in the trial libraries used by the SINDy models.

On the other hand, when a higher fidelity drag model is used—one which contains terms missing from and poorly approximated by the library functions—SINDy struggles to identify coherent dynamics. Using the drag model given in (2.10) and (2.11) (see Section 2.5.7) to simulate a falling ball, SINDy learns the governing equation

$$\dot{v} = -6.345$$

for a "large" threshold value (0.1). The constant acceleration term is shifted away from the true value to compensate for the drag. For a small "threshold" (0.004), SINDy learns the following model

$$\dot{v} = -9.810 - 0.005v + 0.17v^2.$$

The constant acceleration is very close to the true value, but there is also a nonphysical positive quadratic term. Without including rational and other more complicated nonlinear functions in the library[4], SINDy lacks the proper building blocks to perfectly reconstruct the behavior of the system. Poor performance can be a signal that some information is not being captured by the library, which is typically chosen based on one's underlying assumptions about the dynamics being studied. In this way SINDy can help reveal discrepancies between the assumed form of the governing equations and reality without necessarily exposing the precise nature of the discrepancy. If one finds that SINDy is producing unreliable models, a possible remedy is to enrich the library of candidate right-hand side functions.

2.5.4 Numerical differentiation

In this section we explore the error introduced by smoothing and numerical differentiation. More specifically in Section 2.5.4 we compare the performance of a few methods of numerical differentiation, in Section 2.5.4 we examine the effects of smoothing on noisy data, and in

[4]Including rational functions in the library introduces additional complications to the SINDy algorithm [67].

Section 2.5.5 we approximate the level of noise present in the actual ball drop data set and use the results of the previous sections to derive estimates for the error in the numerical derivatives used in the main results.

Unless otherwise noted, we worked with a single synthetic trajectory consisting of height measurements generated from an idealized falling object obeying

$$\dot{v} = -9.8 - 0.5v, \qquad v(0) = 0, \qquad x(0) = 40.$$

This particular model was chosen because it is qualitatively similar to the actual trajectories. The measurements are taken at a rate of 15 per second to further imitate the experimental setup. We then add various amounts of Gaussian noise to the measurements. In the plots that follow "noise level" refers to the standard deviation of the noise added. Figure 2.12 shows the trajectory with various amounts of noise. Note that even a noise level of 0.1 is almost indistinguishable from the true trajectory.

Differentiation method comparison

We evaluate four numerical differentiation variants: two (first order) forward difference methods and two (second order) centered difference methods. For one method of each order we apply Savitzgy-Golay smoothing before performing computing the derivative. For the remaining two methods (one first order and one second order) we do not smooth the data before taking the derivative. We use a window size of 35 when performing smoothing. We compute both the first derivative (velocity) and second derivative (acceleration) of the simulated trajectory since the associated differential equation is second order.

Figure 2.13 summarizes our results. There are a few observations to be made:

- The smoothed versions of the methods exhibit much better performance than the unsmoothed variants as the noise level increases.

- Once enough noise is introduced, all the methods considered see their accuracy degraded roughly linearly with the noise level.

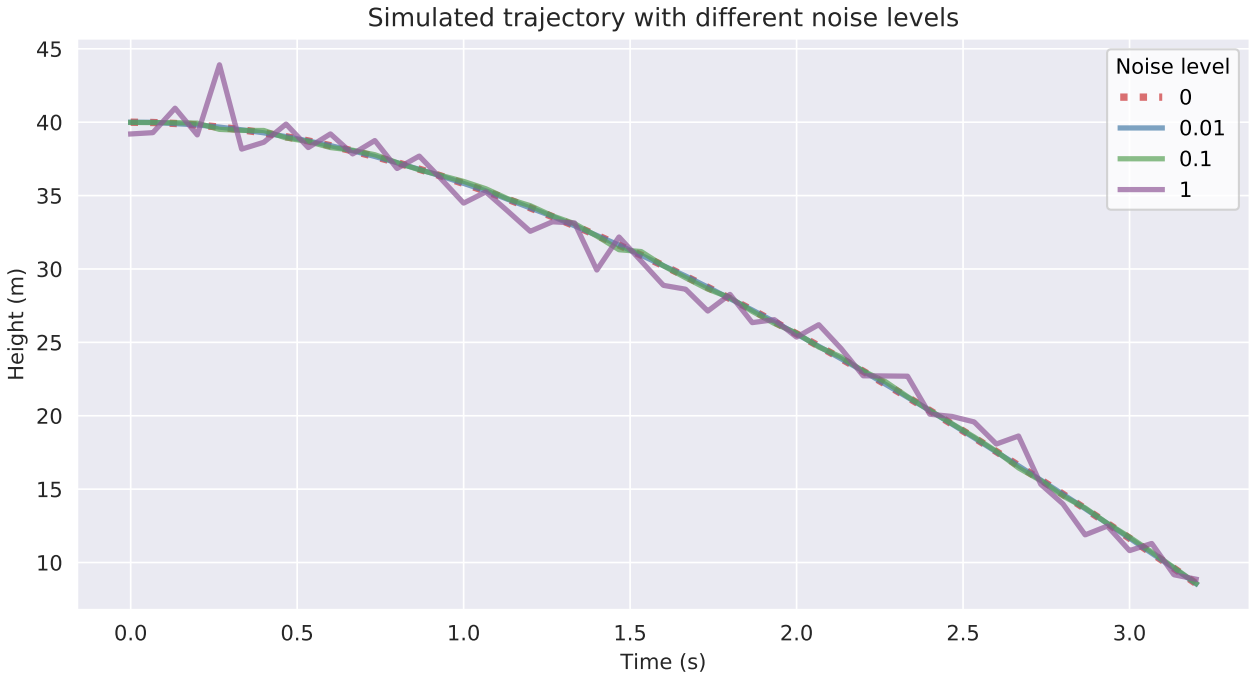


Figure 2.12: The simulated trajectory used for our numerical differentiation and smoothing experiments with varying amounts of noise added.

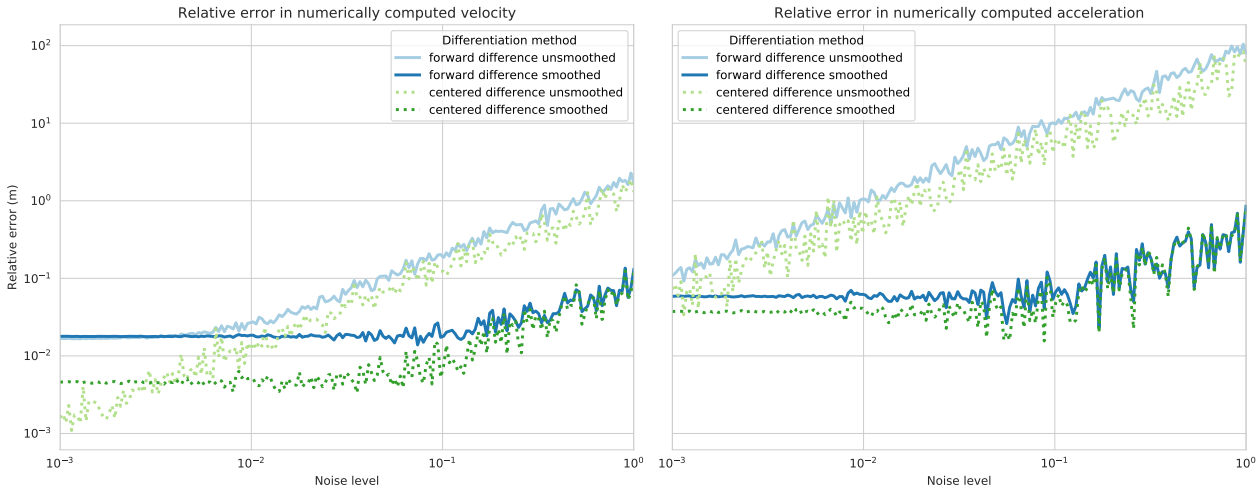


Figure 2.13: Left: Relative error in the *first* derivative of the trajectory computed using four differentiation methods with varying amounts of noise. Right: Relative error in the *second* derivative of the trajectory computed using four differentiation methods with varying amounts of noise.

- The error levels are higher for the approximate acceleration than for the velocity. This makes sense since some error is introduced in computing the velocity and the velocity is needed to compute the acceleration.

- At a low enough noise level there tends to be little difference between the smoothed and unsmoothed versions of each method. The unsmoothed centered difference method outperforms its smoothed counterpart in computing the velocity of relatively clean data.

A conclusion we can draw from this analysis is that the smoothed centered difference method provides the best performance over most levels of noise for both the first and second derivatives.

Smoothing

In the previous experiment we used a fixed window length without justifying our choice. In this section we fix the differentiation method used — centered difference with smoothing — and vary the window length. A larger window means that more points are considered when performing smoothing. The window length roughly translates to smoothness; the larger the window the smoother the result.

Figure 2.14 plots how the error in numerically computed derivatives is affected by the size of the smoothing window used as a function of noise. For small noise levels, larger smoothing windows hurt the method; overly aggressive smoothing throws out some useful information. As the noise levels increase the opposite is true; larger amounts of smoothing are needed to keep the excessive noise at bay. Which window length we should actually use will depend on the noise level we suspect is present in the real-world data set.

2.5.5 Estimating noise in measurement data

In order to infer the amount of noise in the measured ball trajectories it will prove useful to know roughly how much the act of smoothing a trajectory changes the underlying data.

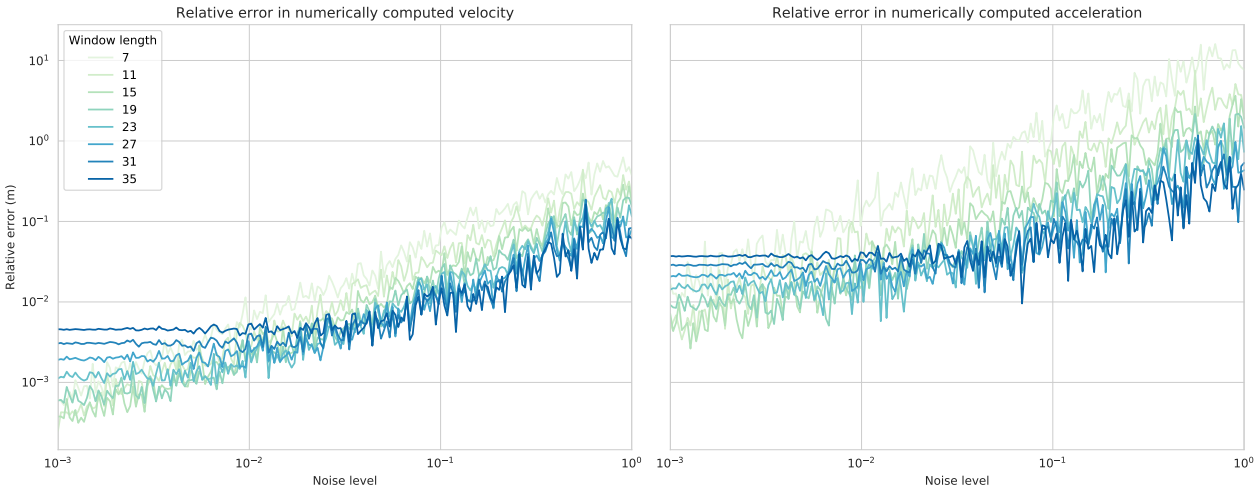


Figure 2.14: The effects of the size of the smoothing window on derivative approximation error. Left: Relative error in the *first* derivative of the trajectory computed using a smoothed centered difference method with different smoothing window sizes. Right: Relative error in the *second* derivative of the trajectory computed using a smoothed centered difference method with different smoothing window sizes.

To this end we perform a similar experiment as in the previous section, but with the height data itself. That is to say we apply the same smoothing operation used before to the height data and measure the relative difference between the smoothed and original data. For completeness, we carry out this experiment for multiple window lengths.

Our results are shown in Figure 2.15. As one would expect, smaller windows produce smoothed trajectories that are closer to their unsmoothed counterparts, but only slightly so. Large smoothing windows have the most pronounced effects when the noise levels are very small and smoothing is unnecessary. For higher noise levels, changes in window size affect the relative difference very little.

Based on these results we elect to use the largest window size tested in our experiments for the results of Section 2.4. SINDy depends heavily on accurate numerical derivatives. For large amounts of noise, a larger window size is necessary for numerical differentiation to work well. We are only penalized for using a large window (in the sense that we greatly modify the original data when we perform smoothing) if the underlying noise level is below about

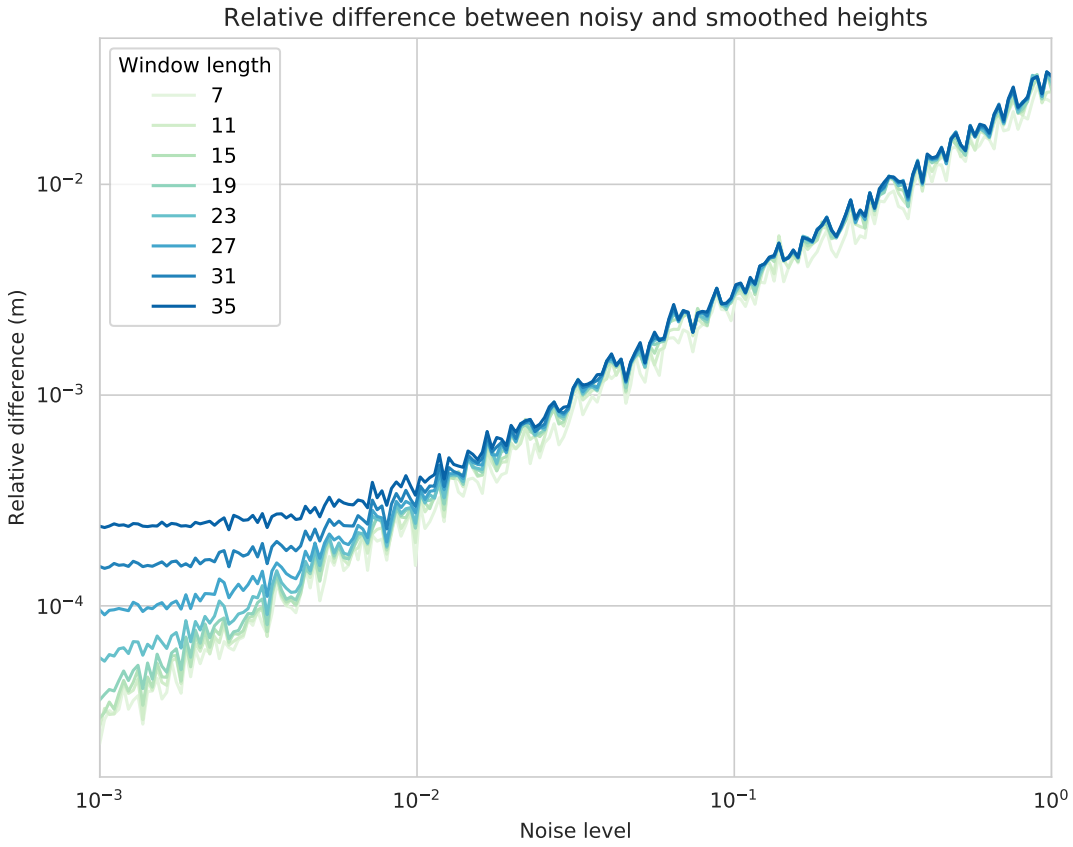


Figure 2.15: The relative difference between noisy trajectories and their smoothed versions for different length smoothing windows.

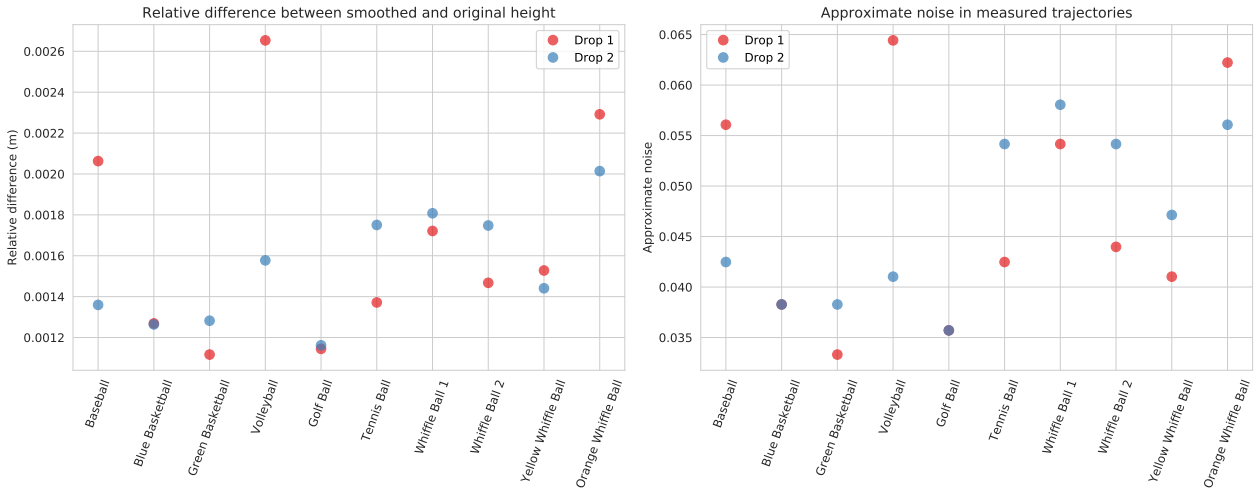


Figure 2.16: Left: Relative difference between the smoothed and unsmoothed falling ball trajectories for both drops (window length = 35). Right: Approximate noise levels present in the ball drop measurements.

10^{-2}.

Next we turn to the task of actually estimating the noise present in the drop data. To accomplish this we apply smoothing with a window length of 35 to each falling ball trajectory, then measure the relative difference between the smoothed and unsmoothed versions. Finally we compare this relative difference with Figure 2.15 to obtain an approximation to the noise level. Figure 2.16 visualizes the relative differences along with the inferred noise levels for each ball drop. The estimated noise levels are all between 0.035 and 0.065. Comparing these results with Figure 2.13, we can deduce that the numerically computed velocity and acceleration vectors have relative errors of order 10^{-3} and 10^{-2}, respectively. It should be noted that we use the ℓ^2 norm when computing relative error. If the ℓ^∞ norm is used instead, the relative errors increase marginally.

2.5.6 Effect of varying sparsity parameter

In this section we provide a representative example of the that are produced when the sparsity threshold parameter is varied. We consider both the regularized and unregularized

SINDy variants. The effect of varying the sparsity parameter is similar In both cases. For large values of the parameter (reflecting a strong preference toward a very sparse solution), no terms are deemed "important" enough to be retained and the trivial model is returned. As the threshold is continuously decreased, a small number of terms will be selected for a range of threshold values. Eventually, when the threshold becomes small enough, suddenly there will be a noticeable jump in the number of terms in the models returned by SINDy. This is typically when one can assume that the sparsity parameter has been made too small. We demonstrate this pattern in Tables 2.3 and 2.4, which give the learned equations for unregularized and unregularized SINDy, respectively, for a variety of sparsity thresholds. The parameter values were chosen to be close to values at which the number of terms in the resulting models changed. For the experiments carried out in Section 2.4 we chose thresholds which were slightly larger than the values at which the jumps in numbers of model terms occurred.

Threshold	**Equation**
10	$v' = 0$
2	$v' = -7.6344$
0.1	$v' = -14.865 + 0.1084x - 0.2914v$
0.005	$v' = -6.1068 - 0.0717x + 0.0880v - 0.0059xv$
0.0045	$v' = -2.9116 - 0.1388x + 0.0861v - 0.0061xv - 0.0048v^2$
0.0035	$v' = 14.7998 - 0.6964x + 0.7036v - 0.0182xv + 0.0039x^2 - 0.0065v^2$
0.002	$v' = 45.4998 - 1.4749x + 2.4829v - 0.0559xv + 0.0067x^2 - 0.0393v^2 - 0.0021v^3$

Table 2.3: Models learned by unregularized SINDy for different threshold parameters (tennis ball, drop one).

In cases where one wishes to perform automatic parameter tuning, cross-validation should allow for one to choose an appropriate sparsity parameter value. Models that are overly sparse

Threshold	Equation
70	$v' = 0$
65	$v' = -6.9$
2	$v' = -8.3 - 0.1v$
0.2	$v' = -15.6 + 0.1x - 0.3v$
0.14	$v' = -2.0 - 0.1x + 0.4v - 0.01xv$
0.1	$v' = 1.5 - 0.2x + 0.4v - 0.01xv + 0.001v^2$
0.05	$v' = -13.1 + 0.2x - 0.6v + 0.008xv - 0.003x^2 - 0.009v^2$
0.02	$v' = 21.7 - 0.7x + 1.7v - 0.04xv + 0.002x^2 - 0.05v^2 - 0.003v^3$
0.01	$v' = -35.2 + 1.1x - 4.7v + 0.13xv - 0.01x^2 - 0.2v^2 - 0.0009x^2v + 0.003xv^2 - 0.001v^3$
0.005	$v' = 9.9 - 0.8x - 0.8v - 0.04xv + 0.02x^2 - 0.005v^2 + 0.0004x^2v - 0.0004xv^2 - 0.0002x^3 - 0.0003v^3$

Table 2.4: Models learned by regularized SINDy for different threshold parameters (all balls, drop one).

(those which have too large a sparsity parameter) will be too simple to accurately predict unseen data and models that are not sparse enough (those which have too small a sparsity parameter) will overfit the training data and will generalize poorly. Poor performance on the holdout/validation/test set should catch both overfit and underfit models.

One troublesome case that is possible with SINDy is when the model jumps directly from underfitting to overfitting as the sparsity parameter is varied. This could occur for a number of reasons, but the primary suspects are typically:

- The library is not rich enough to properly capture the dynamics (i.e. one or more of the terms in the "true" underlying dynamical system are not present in the library being used by SINDy).

- The library is too rich. If too many functions are included in the library then the system solved by SINDy can become ill-conditioned, leading to unpredictable results.

- The data are not described by a dynamical system. If this is the case then SINDy is not an appropriate tool.

- Data are too noisy. Recovering a sparse solution from extremely noisy data may not be possible.

2.5.7 Realistic falling ball simulations

In the main work we simulate falling balls with *constant* drag proportional to the balls' velocities. However, in reality, the drag varies nonlinearly with Reynolds number, which it itself a function of velocity. In this section we discuss what the two SINDy models are able to learn when a more complicated, but physically accurate model is used to construct the synthetic ball drops. Specifically, to simulate falling spheres, we numerically solve the following initial value problem for 49 time steps of length 1/15 seconds (to mimic the real-world experiments)

$$m\dot{v} = mg + \frac{1}{2}\rho v^2 A C_D, \qquad v(0) = 0 \tag{2.10}$$

where m is the mass of the ball, ρ is the density of air, $A = \pi r^2$ is the cross-sectional area of the sphere, r is the radius of the sphere, and C_D is the Reynolds number dependent drag coefficient. We use $\rho = 1.211$ kg/m^3 (the density of air at sea level with a temperature of 65 degrees Fahrenheit). For C_D we use the following approximation which is based on experimental measurements, recommended in [20]:

$$\frac{24}{Re}\left(1 + 0.150 Re^{0.681}\right) + \frac{0.407}{1 + \frac{8710}{Re}}. \tag{2.11}$$

This approximation is valid for $Re < 2 \times 10^5$, just before the so-called "drag crisis" when C_D drops suddenly. We do not attempt to reproduce the behavior of the drag coefficient during and after the drag crisis, only before it. This model also assumes that the spheres

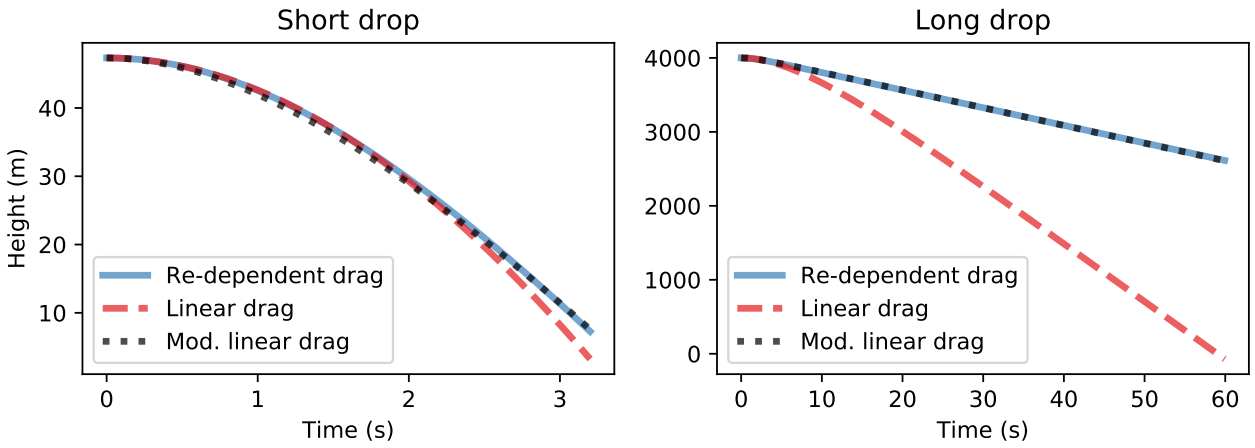


Figure 2.17: A comparison of the simulated trajectories of a tennis ball using a Reynolds number-dependent drag force from (2.10) and (2.11) (solid) and two different constant linear drags (dashed and dotted). On the left we show a short drop similar to the physical experiments and on the right we have simulated a longer drop lasting a full 60 seconds. For the Reynolds number-dependent drag model we used the mass and diameter of the actual tennis ball. For the first linear drag model we used constant gravitational acceleration and a drag coefficient of -0.125 (the average of the two drag coefficients selected by SINDy in the real-world experiments). The modified linear drag model involved constant acceleration of -12.7 m/s^2 and a drag coefficient of -0.53. No noise was added.

are smooth. In Figure 2.17 we compare the simulated trajectories for a tennis ball using both the drag force from (2.10) and (2.11) and the linear drag model presented in the main work. Note that the trajectories initially agree, but as the ball reaches higher velocities and Reynolds numbers, the more complicated Re-dependent model predicts a larger drag force. The difference between these two models becomes clear when the simulations are run for longer amounts of time (900 time steps). The ball effected by linear drag reaches a much faster terminal velocity compared with the other ball. Figure 2.17 also shows how a linear drag model with a larger drag coefficient (and larger constant acceleration) can mimic the Re-dependent model.

As before we simulate five hypothetical balls falling for 49 time steps of duration 1/15 seconds, each with a different mass and radius. The masses and radii were selected to match a subset of the balls in the real-world data set. Table 2.5 gives the characteristics of each

simulated ball. Varying amounts of noise are then added to the artificial measurement data. Finally, we apply the unregularized and group variants of SINDy. The coefficients learned by the two methods are shown in Figure 2.18.

Simulated ball	**Real ball**	**Radius (m)**	**Mass (kg)**
Ball 1	Golf Ball	0.022	0.0454
Ball 2	Tennis Ball	0.033	0.0567
Ball 3	Whiffle Ball 1	0.036	0.0283
Ball 4	Baseball	0.035	0.1417
Ball 5	Blue Basketball	0.119	0.5103

Table 2.5: Properties of the simulated balls and the real balls after which they were modeled.

The unregularized SINDy models exhibit better performance here than in the linear drag case in the sense that they tend not to pick up extraneous terms such as x until relatively high noise levels are present. Notably, many of the models include constant acceleration and linear drag terms. The group sparsity methods perform similarly as before. For low noise levels it detects constant acceleration and linear, but not quadratic drag. As additional noise is introduced, the models erroneously adopt a term proportional to ball height. It should be noted that, in these simulations, the factor multiplying v^2 in 2.10, namely $\frac{1}{2m}\rho A C_D$, does not exceed 0.08, except very early in the balls' trajectories when v is small and v^2 even smaller. The consequence of this observation is that even if this factor were constant with respect to velocity, SINDy and other model discovery methods would have a difficult time accurately detecting it because it is so small relative to the other effects present in the experiment. It should be noted that even if the number of measurements is expanded by increasing the duration of the simulations, SINDy tends to adjust the constant acceleration and linear drag coefficients to match the data rather than adopting a quadratic drag term. Figure 2.17 demonstrates just how closely linear drag can mimic quadratic drag as a ball approaches terminal velocity. If the amount of data is increased by instead collecting more

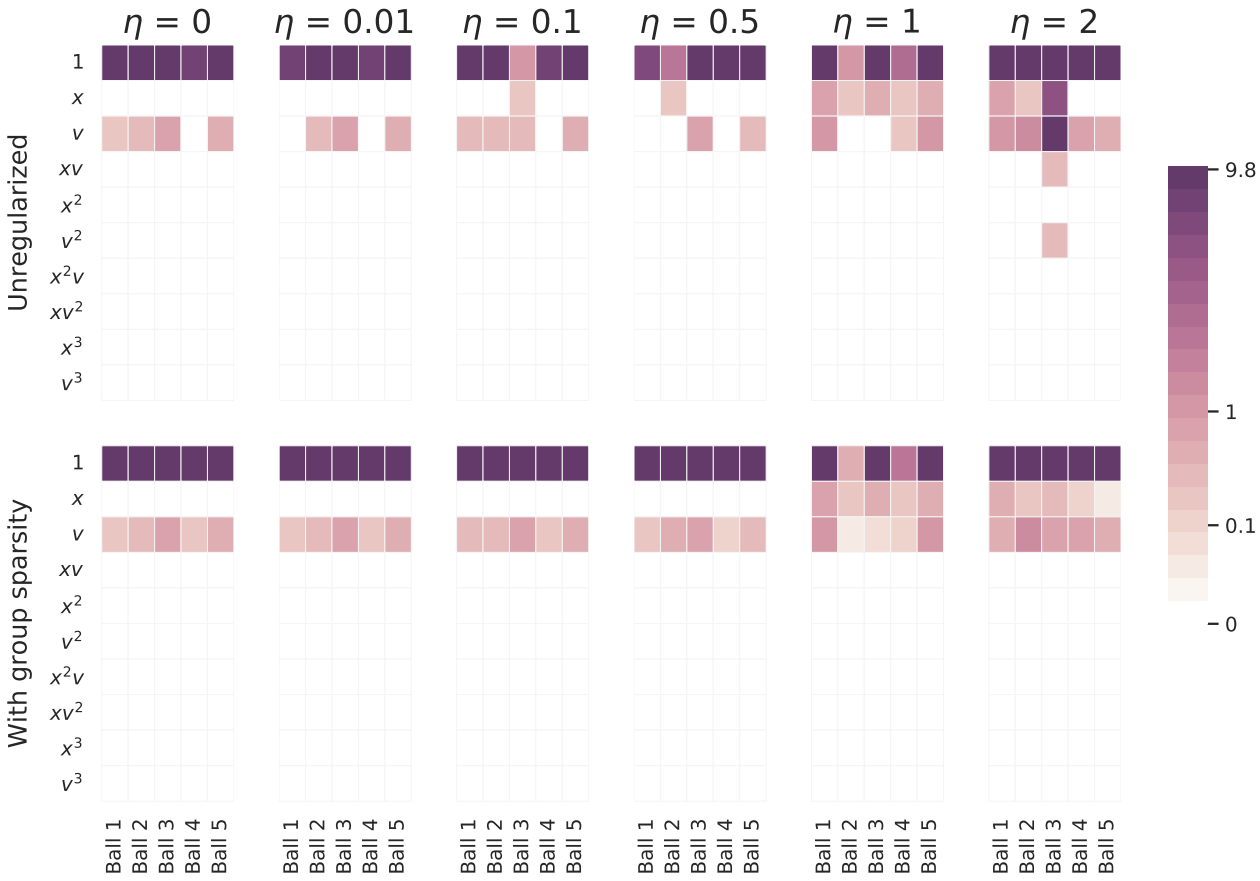


Figure 2.18: A comparison of coefficients of the models inferred from the simulated falling balls. The top row shows the coefficients learned with the standard SINDy algorithm and the bottom row the coefficients learned with the group sparsity method. η indicates the amount of noise added to the simulated ball drops. The standard approach used a sparsity parameter of 0.05 and the group sparsity method used a value of 0.3. The balls' trajectories were simulated using equation (2.10).

measurements over a shorter time span[5] we saw no improvements in the ability of the model to detect a quadratic drag term. Similarly, SINDy accounted for increases in the density of the fluid through which the balls fall by adjusting the gravitational constant and linear drag term.

2.6 Discussion and Conclusions

In this work, we have revisited the classic problem of modeling the motion of falling objects in the context of modern machine learning, sparse optimization, and model selection. In particular, we develop data-driven models from experimental position measurements for several falling spheres of different size, mass, roughness, and porosity. Based on this data, a hierarchy of models are selected via sparse regression in a library of candidate functions that may explain the observed acceleration behavior. We find that models developed for individual ball-drop trajectories tend to overfit the data, with all models including a spurious height-dependent force and lower-density balls resulting in additional spurious terms. Next, we impose the assumption that all balls must be governed by the same basic model terms, perhaps with different coefficients, by considering all trajectories simultaneously and selecting models via group sparsity. These models are all parsimonious, with only two dominant terms, and they tend to generalize without overfitting.

Although we often view the motion of falling spheres as a solved problem, the observed data is quite rich, exhibiting a range of behaviors. In fact, a constant gravitational acceleration is not immediately obvious, as the falling motion is strongly affected by complex unsteady fluid drag forces; the data alone would suggest that each ball has its own slightly different gravity constant. It is interesting to note that our group sparsity models include a drag force that is proportional to the velocity, as opposed to the *textbook* model that includes the square of velocity that is predicted for a constant drag coefficient. However, in reality the drag coefficient decreases with velocity, as shown in Fig. 2.1, which may contribute to

[5]We experimented with increasing the sampling rate to 60 measurements per second over 3.33 seconds.

the force being proportional to velocity. Even when a higher fidelity drag model is used—a model containing rational terms missing from and poorly approximated by the polynomial library functions—to collect measurements uncorrupted by noise, SINDy struggles to identify coherent dynamics. In general SINDy may not exhibit optimal performance if not equipped with a library of functions in which dynamics can be represented sparsely. We emphasize that although the learned models tend to fit the data relatively well, it would be a mistake to assume that they would retain their accuracy for Reynolds numbers larger than those present in the training data. In particular we should expect the models to have trouble extrapolating beyond the drag crisis where the dynamics change considerably. This weakness is inherent in virtually all machine learning models; their performance is best when they are applied to data similar to what they have already seen and dubious when applied in novel contexts. That is to say they excel at interpolation, but are often poor extrapolators.

Collecting a richer set of data should enable the development of refined models with more accurate drag physics[6], and this is the subject of future work. In particular, it would be interesting to collect data for spheres falling from greater heights, so that they reach terminal velocity. It would also be interesting to systematically vary the radius, mass, surface roughness, and porosity, for example to determine non-dimensional parameters. Finally, performing similar tests in other fluids, such as water, may also enable the discovery of added mass forces, which are quite small in air. Such a dataset would provide a challenging motivation for future machine learning techniques.

We were able to draw upon previous fluid dynamics research to establish a "ground truth" model against which to compare the models proposed by SINDy. However, in less mature application areas one may not be fortunate enough to have a theory-backed set of reference equations, making it challenging to assess the quality of learned models. Many methods in numerical analysis come equipped with a priori or a posteriori error estimators or convergence results to give one an idea of the size of approximation errors. Similarly,

[6]We note that in order to properly resolve these more complex drag dynamics with SINDy the candidate library would likely need to be enriched.

in statistics goodness of fit estimators exist to help guide practitioners about what type of performance they should expect from various models. A comprehensive study into whether similar techniques could be adopted for application to SINDy would be an interesting topic for future research efforts.

We believe that it is important to draw a parallel between great historical scientific breakthroughs, such as the discovery of a universal gravitational constant, and modern approaches in machine learning. Although computational learning algorithms are becoming increasingly powerful, they face many of the same challenges that human scientists have faced for centuries. These challenges include trade offs between model fidelity and the quality and quantity of data, with inaccurate measurements degrading our ability to disambiguate various physical effects. With noisy data, one can only expect model identification techniques to uncover the dominant, leading-order effects, such as gravity and simple drag; for subtler effects, more accurate measurement data is required. Modern learning architectures are often also prone to overfitting without careful cross-validation and regularization, and models that are both interpretable and generalizable come at a premium. Typically the regularization encodes some basic human assumption, such as sparse regularization, which promotes parsimony in models. More fundamentally, it is not always clear what should be measured, what terms should be modeled, and what parameters should be varied to isolate the effect one wishes to study. Historically, this type of scientific inquiry has been driven by human curiosity and intuition, which will be critical elements if machine intelligence is to advance scientific discovery.

Chapter 3

PYSINDY

This chapter is based on joint work with Kathleen Champion and Markus Quade.

Scientists have long quantified empirical observations by developing mathematical models that characterize the observations, have some measure of interpretability, and are capable of making predictions. Dynamical systems models in particular have been widely used to study, explain, and predict system behavior in a wide range of application areas, with examples ranging from Newton's laws of classical mechanics to the Michaelis-Menten kinetics for modeling enzyme kinetics. While governing laws and equations were traditionally derived by hand, the current growth of available measurement data and resulting emphasis on data-driven modeling motivates algorithmic approaches for model discovery. A number of such approaches have been developed in recent years and have generated widespread interest, including Eureqa [104], sure independence screening and sparsifying operator (SISSO) [81], and the sparse identification of nonlinear dynamics (SINDy) [21]. Maximizing the impact of these model discovery methods requires tools to make them widely accessible to scientists across domains and at various levels of mathematical expertise.

`PySINDy` is a Python package for the discovery of governing dynamical systems models from data. In particular, `PySINDy` provides tools for applying the sparse identification of nonlinear dynamical systems (SINDy) approach to model discovery [21]. SINDy poses model discovery as a sparse regression problem, where relevant terms in the governing equations are selected from a library of candidate functions. This approach is straightforward to understand and can be readily customized using different sparse regression algorithms or library functions.

The `PySINDy` package is aimed at researchers and practitioners alike, enabling anyone

with access to measurement data to engage in scientific model discovery. The package is designed to be accessible to inexperienced practitioners, while also including options that allow more advanced users to customize it to their needs. A number of popular SINDy variants are implemented, but `PySINDy` is also designed to enable further extensions for research and experimentation.

3.1 Background

`PySINDy` provides an implementation of the SINDy method for discovering governing dynamical systems models. Given data in the form of state measurements $\mathbf{x}(t) \in \mathbb{R}^n$, SINDy seeks a function $\mathbf{f}$ such that

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)). \tag{3.1}$$

Thus $\mathbf{f}$ is a function that describes how the states of the system evolve in time. SINDy sets up the model discovery problem as a sparse regression problem: given some library of candidate basis functions $[\theta_1(\mathbf{x}), \theta_2(\mathbf{x}), \ldots, \theta_\ell(\mathbf{x})]$, the regression seeks the relevant terms in $\mathbf{f}$.

To set up the regression problem, state measurements $\mathbf{x}$ and their time derivatives $\dot{\mathbf{x}}$ are stacked into matrices

$$\mathbf{X} = \begin{pmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{pmatrix}, \quad \dot{\mathbf{X}} = \begin{pmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \cdots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \cdots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \cdots & \dot{x}_n(t_m) \end{pmatrix}.$$

A library matrix $\boldsymbol{\Theta}(\mathbf{X}) = [\theta_1(\mathbf{x}), \theta_2(\mathbf{x}), \ldots, \theta_\ell(\mathbf{x})]$ is formed from the candidate functions, and sparse regression is performed to approximately solve

$$\dot{\mathbf{X}} \approx \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\Xi}, \tag{3.2}$$

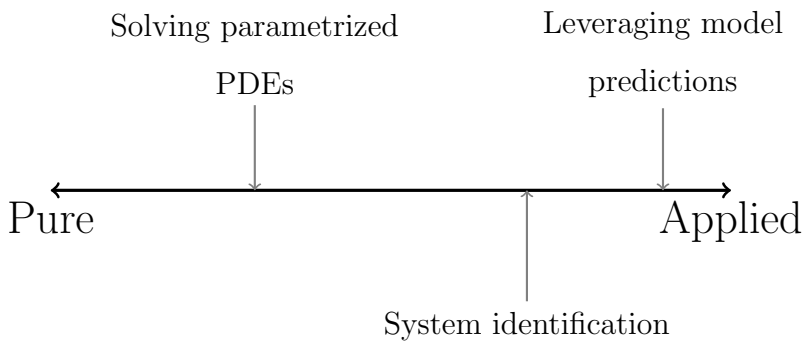
where $\boldsymbol{\Xi}$ is a set of coefficients that determines the terms in $\mathbf{f}$. While the original SINDy formulation solves (3.2) using a sequentially thresholded least squares algorithm [21], this

problem can be solved using any sparse regression algorithm, such as lasso [119] or sparse relaxed regularized regression (SR3) [134].

As we mentioned before, the SINDy method has been widely applied for model identification in applications such as chemical reaction dynamics [47], nonlinear optics [109], thermal fluids [64], plasma convection [34], numerical algorithms [118], and structural modeling [58]. It has also been extended to handle more complex modeling scenarios such as partial differential equations [100, 96], systems with inputs or control [53], corrupt or limited data [120, 102], integral formulations [101, 93], physical constraints [62], tensor representations [40], and stochastic systems [14]. However, there is not a definitive standard implementation or package for applying SINDy. Versions of SINDy have been implemented within larger projects such as `sparsereg` [87], but no specific implementation has emerged as the most widely adopted and most versions implement only a limited set of features. Researchers have thus typically written their own implementations, resulting in duplicated effort and a lack of standardization. This not only makes it more difficult to apply SINDy to scientific data sets, but also makes it more challenging to benchmark extensions to the method against the original and makes such extensions less accessible to end users. This motivates the creation of a dedicated package for SINDy. The `PySINDy` package provides a central codebase where many of the basic SINDy features are implemented, allowing for easy use and standardization. In addition, this makes it straightforward for users to extend the package in a way such that new developments are available to a wider user base.

3.2 Features

The core object in the `PySINDy` package is the `SINDy` model class, which is implemented as a `scikit-learn` estimator. This design choice was made to to ensure the package is simple to use for a wide user base, as many potential users will be familiar with `scikit-learn`. It also expresses the `SINDy` model object at the appropriate level of abstraction so that users can embed it into more complicated pipelines in `scikit-learn`, such as tools for parameter tuning and model selection.



Our implementation of `SINDy` involves three major steps, and, therefore, three primary modeling decisions, namely:

1. which *numerical differentiation* method is used to approximate $\dot{\mathbf{X}}$ from $\mathbf{X}$,
2. which functions make up the *feature library* $\boldsymbol{\Theta}$,
3. and which *sparse regression* algorithm is applied to solve (3.2) (i.e. to find $\boldsymbol{\Xi}$).

The core `SINDy` object was designed to make these three steps as modular as possible. `PySINDy` provides some standard options for each step, while making it easy to swap out one or more for alternate "third-party" algorithms. In particular, at the time of writing, we have implemented the following methods

- Numerical differentiation
  - Finite difference
  - Smoothed finite difference
- Feature library
  - Multivariable polynomials
  - Fourier modes (i.e. trigonometric functions)

  - Custom library (defined by user-supplied functions)
  - Identity library (in case users want to compute $\boldsymbol{\Theta}$ themselves)

- Optimizer
  - Sequentially thresholded least-squares [21]
  - Sparse relaxed regularized regression (SR3) [134]

The GitHub repository housing `PySINDy` includes tutorials in the form of Jupyter notebooks. These tutorials demonstrate the usage of various features of the package and reproduce the examples from the original SINDy paper [21]. We walk through a few examples showing how to use these options in the following section and give tips for choosing among them in Section 3.4.

3.3 Examples

Throughout this section we will use the Lorenz equations (3.3) as the dynamical system we would like to discover.

$$\begin{cases} \dot{x} = -10x - 10y \\ \dot{y} = x(28 - z) - y \\ \dot{z} = xy - \frac{8}{3}z \end{cases} \tag{3.3}$$

In Python, the right-hand side of (3.3) can be expressed as follows.

```
def lorenz(z, t):
    return [
        10 * (z[1] - z[0]),
        z[0] * (28 - z[2]) - z[1],
        z[0] * z[1] - (8 / 3) * z[2]
    ]
```

To construct training data to feed into a SINDy model, we integrate (3.3) by running the following Python commands.

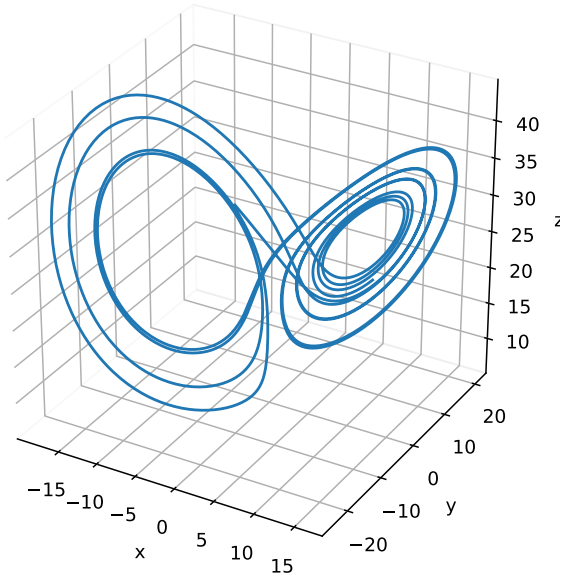


Figure 3.1: Measurement data simulated using the Lorenz equations (3.3).

```
import numpy as np
from scipy.integrate import odeint

dt = 0.002
t = np.arange(0, 10, dt)
x0 = [-8, 8, 27]
x = odeint(lorenz, x0, t)
```

We plot `x` in Figure 3.1. It is important to note that each column of `x` corresponds to a variable and each row to a point in time. All `PySINDy` objects that handle data assume the data is structured this way.

3.3.1 Basic usage

The `pysindy` package is built around the `SINDy` class, which encapsulates all the steps necessary to learn a dynamical system with SINDy and the model itself. To create a SINDy object, fit it to the data (i.e. to infer a dynamical system from the data), and print the

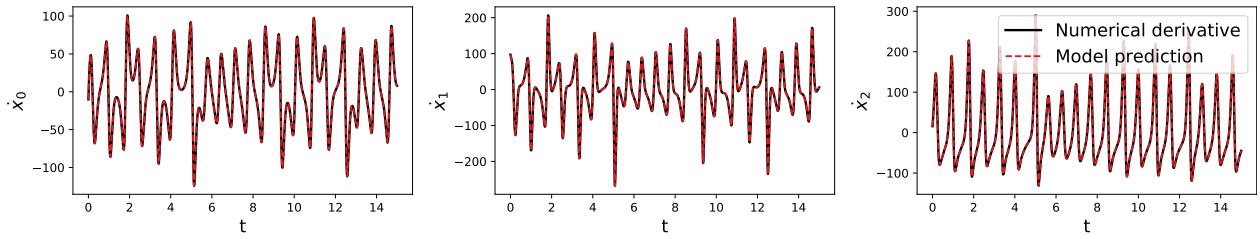


Figure 3.2: Derivatives of variables from the Lorenz equation via numerical differentiation and using a learned SINDy model.

resulting model, we invoke the `SINDy` constructor, the `fit` method, and custom `print()` functions

```
model = ps.SINDy()
model.fit(x, t=dt)
model.print()
```

which generates the following output.

```
x0' = -9.999 x0 + 9.999 x1
x1' = 27.992 x0 + -0.999 x1 + -1.000 x0 x2
x2' = -2.666 x2 + 1.000 x0 x1
```

Once the SINDy object has been fit we can feed in new data and use the learned model to predict the derivatives for each measurement (recall that measurements correspond to rows).

```
t_test = np.arange(0, 15, dt)
x0_test = np.array([8, 7, 15])
x_test = odeint(lorenz, x0_test, t_test)

x_dot_test_computed = model.differentiate(x_test, t=dt)
x_dot_test_predicted = model.predict(x_test)
```

The call `SINDy.differentiate(x_test, t=dt)` applies a numerical differentiation method to `x_test` with time steps of length `dt`. In Figure 3.2 we plot each dimension of `x_dot_test_computed` and `x_dot_test_predicted`.

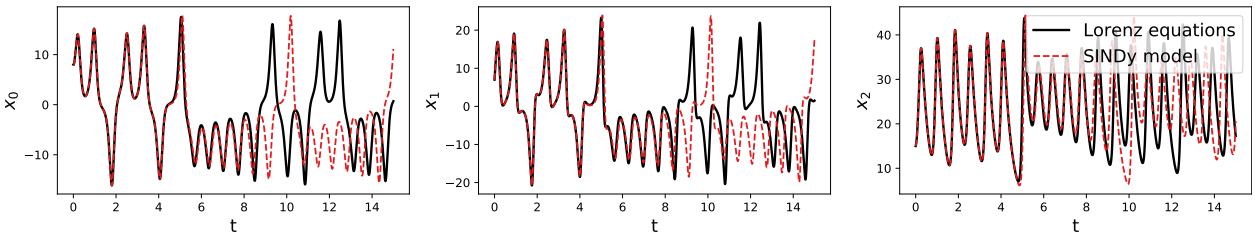


Figure 3.3: Two trajectories starting at the same position evolved forward in time with the exact Lorenz equations (black, solid) and the learned SINDy model (red, dashed).

Rather than predicting derivatives, we will often be interested in using our learned model to evolve initial conditions forward in time using the learned model. The `simulate` function does just that.

```
x_test_sim = model.simulate(x0_test, t_test)
```

Plotting this simulated trajectory against the true trajectory `x_test` yields Figure 3.3. The trajectories agree initially, but because of the chaotic nature of the Lorenz equations, eventually they diverge.

3.3.2 Custom features

Thus far we have relied on the default options of the `SINDy` object, but `PySINDy` comes equipped with multiple alternative built-in methods for differentiation, library building, and optimization. These options are selected by passing corresponding `PySINDy` objects to the `SINDy` constructor via the `differentiation_method`, `feature_library`, and `optimizer` arguments, respectively. Parameters for the differentiation, library, and optimization algorithms are supplied to the objects' constructors. We demonstrate the syntax with the following example

```
differentiation_method = ps.FiniteDifference(order=1)
feature_library = ps.PolynomialLibrary(degree=3, include_bias=False)
optimizer=ps.SR3(threshold=0.1, nu=1, tol=1e-6)

```

```
model = ps.SINDy(
    differentiation_method=differentiation_method,
    feature_library=feature_library,
    optimizer=optimizer,
    feature_names=["x", "y", "z"]
)

model.fit(x, t=dt)
model.print()
```

which prints

```
x' = -10.021 x + 9.993 y
y' = 28.431 x + -1.212 y + -1.008 x z
z' = -2.675 z + 1.000 x y.
```

A number of other built-in options are available. The official documentation[1] and examples[2] provide an exhaustive list.

3.4 Practical tips

In this section we provide pragmatic advice for using `PySINDy` effectively. We discuss potential pitfalls and strategies for overcoming them. We also specify how to incorporate custom methods not implemented natively in `PySINDy`, where applicable. The information presented here is derived from a combination of experience and theoretical considerations.

3.4.1 Numerical differentiation

Numerical differentiation is one of the core components of the SINDy method. Derivatives of measurement variables provide the targets for the sparse regression problem (3.2). If care is not taken in computing these derivatives, the quality of the learned model is likely to suffer.

[1] `https://pysindy.readthedocs.io/en/latest/index.html`

[2] `https://github.com/dynamicslab/pysindy/tree/master/example`

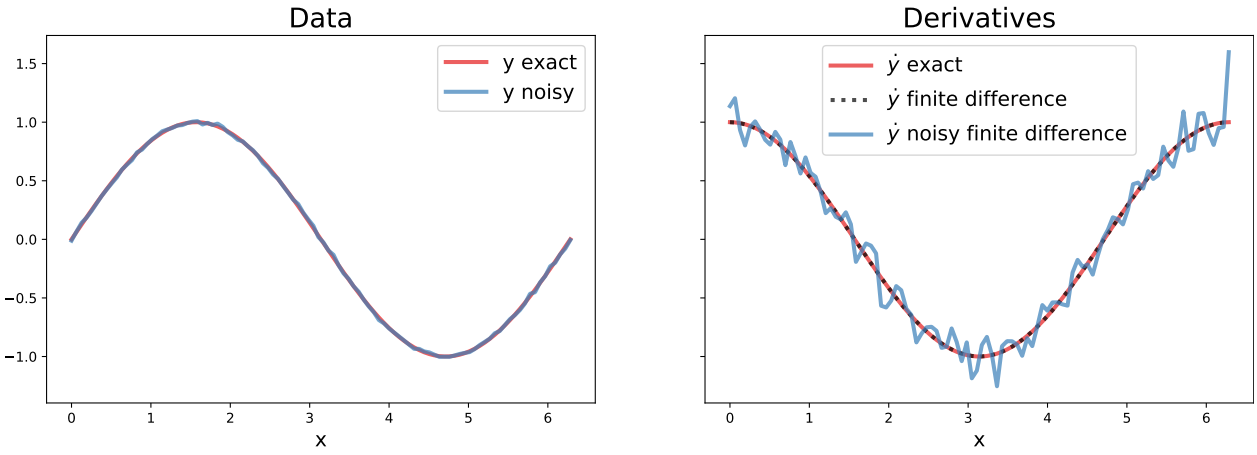


Figure 3.4: A toy example illustrating the effect of noise on derivatives computed with a second order finite difference method. Left: The data to be differentiated; $y = \sin(x)$ with and without a small amount of additive noise (normally distributed with mean 0 and standard deviation 0.01). Right: Derivatives of the data; the exact derivative $\cos(x)$ (blue), the finite difference derivative of the exact data (black, dashed), and the finite difference derivative of the noisy data.

By default, a second order finite difference method is used to differentiate input data. Finite difference methods tend to amplify noise in data. If the data are smooth (at least twice differentiable), then finite difference methods give accurate derivative approximations. When the data are noisy, they give derivative estimates with *more* noise than the original data. Figure 3.4 visualizes the impact of noise on numerical derivatives. Note that even a small amount of noise in the data can produce noticeable degradation in the quality of the numerical derivative.

One way to mitigate the effects of noise is to smooth the measurements before computing derivatives. The `SmoothedFiniteDifference` method can be used for this purpose. A numerical differentiation scheme with total variation regularization has also been proposed [30] and recommended for use in SINDy [21].

Users wishing to employ their own numerical differentiation schemes have two ways of doing so. Derivatives of input measurements can be computed externally with the method of choice then passed directly into the `SINDy.fit` method via the `x_dot` keyword argument.

Alternatively, users can implement their own differentiation methods and pass them into the `SINDy` constructor using the `differentiation_method` argument. In this case, the supplied class need only have implemented a `__call__` method taking two arguments, `x` and `t`.

3.4.2 Library selection

The SINDy method assumes dynamics can be represented as a *sparse* linear combination of library functions. If this assumption is violated, the method is likely to exhibit poor performance. This issue tends to manifest itself as numerous library terms being active, often with weights of vastly different magnitudes.

Typically, prior knowledge of the system of interest and its dynamics should be used to make a judicious choice of basis functions. When such information is unavailable, the default class of library functions, polynomials, are a good place to start as smooth functions have rapidly converging Taylor series. Brunton et al. showed that, equipped with a polynomial library, SINDy can recover the first few terms of the (zero-centered) Taylor series of the true right-hand side function $\mathbf{f}(x)$ [21]. If one has reason to believe the dynamics can be sparsely represented in terms of Chebyshev polynomials rather than monomials, then the library should include Chebyshev polynomials.

`PySINDy` includes the `CustomLibrary` and `IdentityLibrary` objects to allow for flexibility in library functions. When the library should consist of a set of functions that should be applied to each measurement variable (or pair, triplet, etc. of measurement variables) in turn, the `CustomLibrary` class should be used. The `IdentityLibrary` class is the most customizable, but transfers the work of computing library functions over to the user. It expects that all the features one wishes to include in the library have already been computed and are present in `X` before `SINDy.fit` is called. It simply applies the identity map to each variable it is passed It is best suited for situations in which one has very specific instructions for how to apply library functions (e.g. if some of the functions should be applied to only some of the input variables).

As terms are added to the library, the underlying sparse regression problem becomes

increasingly ill-conditioned. Therefore it is recommended to start with a small library whose size is gradually expanded until the desired level of performance is achieved. For the best results, the strength of regularization applied should be increased in proportion to the size of the library to account for the worsening condition number of the resulting linear system.

Users may also choose to implement library classes tailored to their applications. To do so one should have the new class inherit from our `BaseFeatureLibrary` class. See the documentation for which functions the new class is expected to implement.

3.4.3 Optimization

`PySINDy` uses various optimizers to solve a sparse regression problem. For a fixed differentiation method, set of inputs, and candidate library, there is still some variance in the dynamical system identified by SINDY, depending on which optimizer is employed.

The default optimizer in `PySINDy` is the sequentially-thresholded least-squares algorithm (`STLSQ`). In addition to being the method originally proposed for use with SINDy, it involves just one (easily interpretable) hyperparameter and it exhibits good performance across a variety of problems.

The sparse relaxed regularized regression (`SR3`) algorithm can be used when the results of `STLSQ` are unsatisfactory. It involves a few more hyperparameters which can all be tuned for improved accuracy. In particular, the `thresholder` parameter controls the type of regularization that is applied. For optimal results, one may find it useful to experiment with L^0, L^1, and clipped absolute deviation (CAD) regularization. The other hyperparameters can be tuned with cross-validation.

Custom or third party sparse regression methods are also supported. Simply instantiate an instance of the custom object and pass it to the `SINDy` constructor using the `optimizer` keyword. Our implementation is compatible with any of the linear models from Scikit-learn (e.g. `RidgeRegression`, `Lasso`, and `ElasticNet`). See the documentation for a list of methods and attributes a custom optimizer is expected to implement. There you will also find an example where the Scikit-learn `Lasso` object is used to perform sparse regression.

3.4.4 Regularization

Regularization, in this context, is a technique for improving the conditioning of ill-posed problems. Without it, one often obtains highly unstable results with learned parameter values differing substantially for slightly different inputs. SINDy seeks weights that express dynamics as a *sparse* linear combination of library functions. When measurement data is statistically correlated or large libraries are used, this problem can quickly become ill-posed. Though the sparsity constraint is a type of regularization in and of itself, for many problems another form of regularization is needed for SINDy to learn a robust dynamical model.

In some cases regularization can be interpreted as enforcing a prior distribution on the model parameters [11]. Applying strong regularization biases the learned weights *away* from the values that would allow them to best fit the data and *toward* the values preferred by the prior distribution (e.g. L^2 regularization corresponds to a Gaussian prior). Therefore once a sparse set of nonzero coefficients is discovered, our methods apply an extra "unbiasing" step where *unregularized* least-squares is used to find the values of the identified nonzero coefficients. All of our built-in methods use regularization by default.

Some general best practices regarding regularization follow. Most problems will benefit from some amount of regularization. Regularization strength should be increased as the size of the candidate right-hand side library grows. If warnings about ill-conditioned matrices are generated when `SINDy.fit` is called, more regularization may help. We also recommend setting `unbias` to `True` when invoking the `SINDy.fit` method, especially when large amounts of regularization are being applied. Cross-validation can be used to select appropriate regularization parameters for a given problem.

3.5 Extensions

In this section we list potential extensions and enhancements to our SINDy implementation. We provide references for the improvements that are inspired by previously conducted research and the rationale behind the other potential changes.

- **Integral formulation:** We previously discussed how measurement data with too much noise can disrupt the model discovery process and offered smoothing as one possible solution. Another is to work with an integral version of (3.1), as proposed by Schaeffer and McCalla [101]. Where numerical differentiation tends to amplify noise, numerical integration tends to smooth it out. This formulation has been shown to improve the robustness of SINDy to noise.

- **Partial differential equations (PDEs):** While the flexibility of dynamical systems for modeling physical systems is great, there are other types of governing equations which are not currently discoverable using `PySINDy`. Partial differential equations (PDEs) are one such class of models. Multiple approaches for the data-driven discovery of PDEs have been proposed [96, 97, 100], some being direct descendants of SINDy.

- **Ensembles:** Ensembles are a proven class of methods in machine learning for variance reduction (improved model generalizability) at the cost of extra computation. Rather than training a single model, multiple high-variance models are trained and their predictions are averaged together. We think that ideas from ensemble learning could be adapted to improve the performance of SINDy models. Recent work has given hints about possible approaches [98].

- **Constraints:** SINDy has been extended to enforce *physical constraints* during the sparse regression step [62]. When working with physical systems with known conserved quantities, for example, such a method allows one to automatically incorporate this prior information into the model discovery process.

- **Extended libraries:** Choosing the appropriate basis in which to represent dynamics is of critical importance for the successful application of SINDy. Although we currently provide methods allowing users the flexibility to input their own library functions, we aim to make the library construction process even easier by providing a common suite

of tools for the creation and combination of sets of candidate functions. More basis functions could be supported natively, such as nonautonomous terms (those depending explicitly on the dependent variable, time). Taking this idea a step further, specific variables (columns of $\mathbf{X}$) which should not appear on the left-hand side of (3.2) could be identified by the user. This would enable SINDy to include inputs and control variables [53]. Operations acting on one or more libraries could also be implemented. For example, combining libraries via union, intersection, composition, or tensor product could enable the expression of complicated nonlinear dynamics. The ability to apply libraries to only subsets of state variables could help cut down on the computational cost and improve the conditioning of the sparse regression problem solved within SINDy.

Chapter 4

PHYSICS-INFORMED MACHINE LEARNING FOR SENSOR FAULT DETECTION WITH FLIGHT TEST DATA

This chapter is based on joint work with Jared Callahan, Jonathan Jonker, and Sasha Aravkin.

In this chapter we describe a simple *fully automated* approach to sensor fault detection in systems with underlying physics. The proposed method identifies an approximation to a linear time-invariant system describing the evolution of measurements of interest from a time series of "typical" behavior. Given additional data from related sensors, a Kalman observer is used to maintain a separate real-time estimate of the measurement of interest. Sustained deviation between the measurements and estimate indicates anomalous behavior. A decision tree, informed by other sensor values, is used to determine the amount of deviation required to constitute a sensor fault. We study the efficacy of the method by applying it to three test systems exhibiting various types of sensor faults: commercial flight test data, an unsteady aerodynamics model, and a model for longitudinal flight dynamics forced by atmospheric turbulence. In the latter two cases we explore fault detection for several prototypical failure modes. The combination of a learned dynamical model with the automated decision tree accurately detects sensor faults in each case.

Sensor fault detection and isolation is an important problem in many fields of engineering. Although catastrophic failures may be obvious, more insidious fault modes such as slow drift and low-frequency oscillations tend to be much more difficult to detect; in practice anomaly identification often relies on engineer expertise. In the context of aircraft dynamics, redundant measurements are typically taken for critical quantities; a degree of robustness to faults is therefore built in via a voting or weighted averaging sensor fusion scheme [4].

This is not always the case in flight test scenarios, where enough data is collected that it is impractical to either automatically detect faults via redundancy or to manually monitor for them. Fast, automatic anomaly detection therefore stands to reduced flight test time, resulting in significant cost savings. Even normal service with redundant sensing currently requires structural designs that can withstand flight control signals based on faulty measurements. Robust and guaranteed fault detection/isolation could lead to designs with improved performance and reduced environmental impact [44].

Automated model-based fault detection has been well-studied for linear systems [70, 131, 124], but the general problem remains open for nonlinear dynamics [113]. One approach when a nonlinear model is available is to estimate the state with an extended Kalman filter; consistent discrepancies between the model and estimates that fall outside of the range of process noise may indicate anomalous sensor behavior [60, 125, 41]. This has been applied to flight dynamics models for identification [45] and automatic isolation [38] of anomalous sensors.

If a physics-based model is unavailable, data-driven methods offer an attractive alternative to detect faults, for example by training neural networks [78, 92, 77]. Although in recent years neural networks have had impressive successes in fields such as image recognition, their behavior tends to be unpredictable outside of training conditions: the regime that is arguably most important for investigating anomalous behavior. For this reason we opt for methods which are more amenable to interpretation and analysis, and which offer the possibility of theoretical guarantees (under assumptions on the dynamics, process noise distribution, etc.). An alternative is data-driven system identification [50, 51, 105, 103, 121, 23]. These methods can identify a dynamical model that is suitable for filter-based state estimation and fault detection [112, 113].

In this chapter we build on previous work in data-driven model identification and anomaly detection, in particular [38] and [113], by combining optimal estimation and system identification with modern machine learning methods. We verify that the proposed approach is applicable to both strongly nonlinear dynamics and correlated, non-Gaussian process noise,

and demonstrate fault detection on data from flight tests. A core component of the algorithm is a simple Kalman observer which is used to predict future sensor values based on current measurements. When the difference between this prediction and the observed values differs by enough over a period of time, a sensor fault is flagged. The Kalman filter requires a model of the underlying dynamics from which to estimate future states. This model is learned from data via the dynamic mode decomposition with time-delays. A decision tree is then used to determine the threshold for what constitutes "too big" of a gap between prediction and observation. The model is fully automatic; one need only specify a set of labeled training data and it will learn both a model for the dynamics present in the data and a set of rules for detecting sensor failures.

The key advantages of this approach are that it is fully data-driven (no model is needed for the dynamics underlying the system being monitored, only measurements) and automated, and it can readily support a large number of measurements/features. In our view, its primary disadvantage is that it is a supervised method, meaning that one must supply labeled data to train the model. The techniques out of which the model is built are all fairly general, granting it a large amount of flexibility while simultaneously restricting its accuracy for some specific applications. In domains in which underlying physical dynamics are well-understood, one may get more mileage from a more specialized method.

The chapter is structured as follows. In Section 4.1 we give an overview of the mathematical background underlying our approach before describing the proposed method itself in Section 4.2. Section 4.3 discusses three example flight applications: one real-world flight test dataset and two simulated examples. We conclude with Section 4.4 which provides some final thoughts.

4.1 Background

Here we provide a mathematical foundation for the proposed method. Sections 4.1.1 and 4.1.2 describe Kalman filters and their application to anomaly detection. A method for constructing a linear time invariant (LTI) model for use in a Kalman filter is discussed in

Sections 4.1.3 and 4.1.4. Section 4.2.1 explains how the LTI model can be combined with a Kalman filter to build a robust detection algorithm. A brief discussion of decision trees, the final ingredient in our method, is given in Section 4.1.6.

4.1.1 System identification and Kalman filtering

Since their development in the 1960's [54], various forms of Kalman filters have proven useful in fields ranging from robotics to weather prediction. The filter described in this section is a simple form of this powerful tool, but is nonetheless effective in many test problems. The method is essentially a simplification of those proposed in [41, 45, 38, 113], and references therein. It is thus potentially extensible to more complex detection and estimation problems, including those with underlying physical systems exhibiting strongly nonlinear dynamics.

Although application of Kalman filters to fault detection has been proposed since the 1970's, until recently an existing model was necessary for the method. Considering the scale of sensing in flight test applications, developing independent predictive models for the various sensors could be prohibitive. However, a recent development suggested in [113] was to identify a linear time-invariant (LTI) model that estimates the relationship between measurements using the dynamic mode decomposition (DMD) algorithm. DMD is a powerful method originally developed in the fluids community to study spatio-temporal coherence in high-dimensional numerical and experimental fluid flow data [103, 94, 121]. It has since found applications ranging from neuroscience to epidemiology [56]. The method is designed to efficiently extract dominant patterns from very large data sets and automatically uses correlations in the data for improved robustness.

A simple linear model identified with this algorithm may not be accurate enough to account for the complex interactions in the aircraft system. However, although Kalman filters are often used in full-state estimation, if the goal is not estimation but fault detection, the dynamic model does not need to be particularly accurate in order to capture anomalous behavior, as the results below demonstrate. The first step of the procedure is therefore to identify a linear predictive model from a time series of "typical" measurements. Online, this

DMD model is used to maintain a Kalman filtered estimate of the measurement of interest. The variance between the estimate and the actual measurement is monitored, and persistent deviations signal anomalous behavior.

4.1.2 Anomaly detection with Kalman filters

The following method of Section 4.2.1 is a simplified version of proposed fault detection algorithms. See for example [41, 45, 38, 113] for descriptions of more sophisticated methods.

Let the vector of measurements we wish to monitor at discrete time step k be denoted by $\mathbf{x}_k \in \mathbb{R}^n$. If we have access to a set of related (but not necessarily redundant) measurements, denote these by $\mathbf{y}_k \in \mathbb{R}^p$. For the sensor fault detection examples, x, a scalar, is the sensor measurement, and $\mathbf{y}$ is a vector of other relevant measurements. Assume we have an LTI model (identified either by dynamic mode decomposition or some other procedure) that predicts the next measurement $\mathbf{x}_{k+1}$, given current information $\mathbf{x}_k$ and $\mathbf{y}_k$:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{y}_k$$

In other words, we treat the related measurements as exogenous inputs in the model. There does not need to be a direct causal relationship in the sense that actuation is usually taken in control theory; these measurements should just help to predict the next measurement of interest. Again, this model may be fairly inaccurate. The use of exogenous inputs is designed to stabilize the model and help to detect drift, high frequency noise, etc. This model may alternately be viewed as a linear regression predicting the next measurement. The simplest Kalman filter is a separate LTI "observer" system that maintains an estimate $\hat{\mathbf{x}}$ of the measurement of interest:

$$\hat{\mathbf{x}}_{k+1} = \mathbf{A}\hat{\mathbf{x}}_k + \mathbf{B}\mathbf{y}_k + \mathbf{K}(\mathbf{x}_k - \hat{\mathbf{x}}_k) \tag{4.1}$$

The new term in this equation is the innovation $(\mathbf{x} - \hat{\mathbf{x}})$ which acts as feedback to stabilize the estimate. In general the Kalman gain K is a matrix, which can be chosen for optimal

convergence of the estimate to the true state, given knowledge of the sensor and process noise covariances. In this case K is a scalar which only acts to smooth the estimated state. The method is relatively insensitive to the choice of Kalman gain, and values from 0.1 to 0.001 were tested with similar performance.

As suggested in for instance [70, 45], a moving average of the innovation covariance can be used to identify anomalous behavior:

$$V_k = \frac{1}{N} \sum_{i=k-N}^{k} (\mathbf{x}_i - \hat{\mathbf{x}}_i)(\mathbf{x}_i - \hat{\mathbf{x}}_i)^\top \tag{4.2}$$

Intuitively, provided the LTI model captures the important correlations between sensors, when a measurement persistently yields anomalous behavior this term will remain large. Even if the model is not particularly accurate, exogenous inputs can still lead to an estimate that reflects anomalous behavior. It is worth emphasizing, however, that without significant additional validation there is no reason to believe that the Kalman filtered estimate is accurate.

For example, Figure 4.1 demonstrates the Kalman filter-based detection of a sensor failure in the real world data set (see Section 4.3.1). An anomaly can be flagged when the innovation covariance exceeds some threshold, which may itself be selected in an automated fashion. Although the filtered estimate is often inconsistent with both the faulty sensor and a redundant, working sensor, before and after the faulty sensor fails, the innovation covariance only grows significantly after the failure. In other words, the filtered estimate does not need to be accurate in order to be an accurate predictor of sensor faults.

4.1.3 Identifying a linear time-invariant model

The method described above requires a reasonably accurate model of the dynamics of the measurement of interest, possibly including the relationship between this measurement and the output of related sensors. In the simplest case (presented here) this could be a linear time-invariant (LTI) system, although application to a nonlinear model is possible using

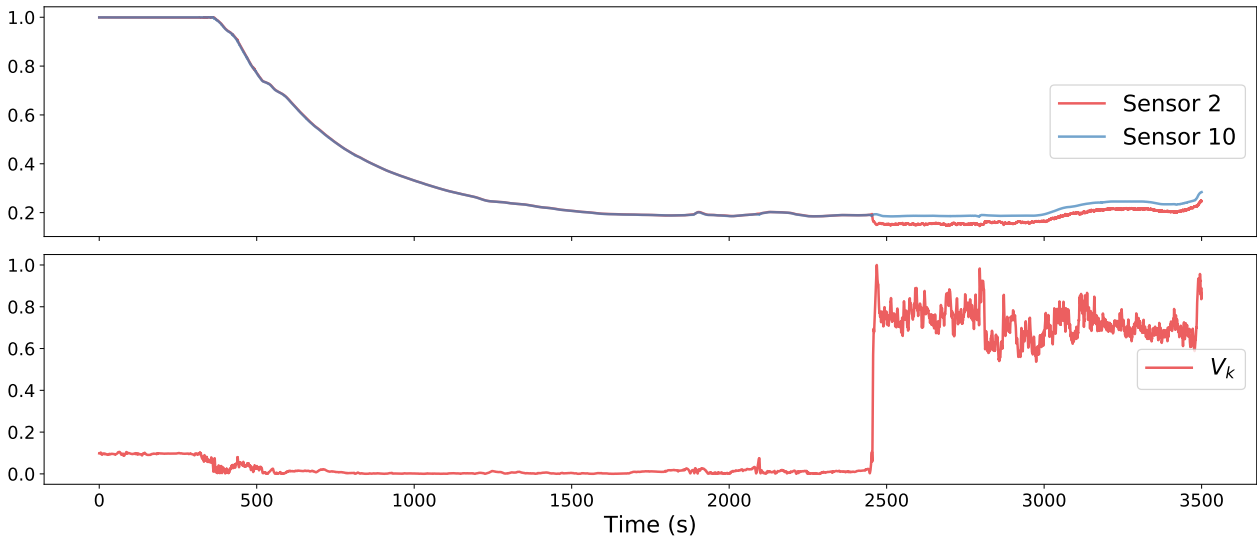


Figure 4.1: Anomaly detection for the flight test dataset of Section 4.3.1 with the Kalman filter. Top: measurements from two redundant sensors. Just before 2500 seconds, sensor 2 breaks and begins giving erratic readings. Bottom: a moving average, V_k, of the covariance term (see 4.2). Note that V_k remains negligible until the sensor failure event leads to persistent anomalous measurements relative to the Kalman filter.

an Extended Kalman Filter (EKF) [41, 45, 38] or optimization-based nonlinear Kalman smoothing approaches [5, 6].

Although obtaining a model in general can be a labor-intensive and problem-specific task, recent developments in system identification have enabled a range of straightforward, efficient model estimation tools. The method presented here uses one such system identification algorithm, called dynamic mode decomposition (DMD). DMD was originally developed in the fluid dynamics community as a method to extract coherent spatio-temporal structures from complex, high-dimensional data [94, 103, 121]. As such, it is designed to take advantage of correlations in the data and reduce the underlying dimensionality of the model. Although there have been many theoretical and numerical refinements of DMD proposed (see e.g. [57, 85, 49, 129, 130]), a simple formulation of the method is as follows:

Suppose we have a series of measurements $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{m+1}\}$ and we assume that these

are related by approximately linear dynamics, i.e.

$$\mathbf{x}_{k+1} \approx \mathbf{A}\mathbf{x}_k.$$

If we arrange the measurements in a time-shifted pair of matrices $\mathbf{X}$ and $\mathbf{X}'$ so that

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \\ | & | & & | \end{bmatrix}, \qquad \mathbf{X}' = \begin{bmatrix} | & | & & | \\ \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_{m+1} \\ | & | & & | \end{bmatrix},$$

these matrices are related by

$$\mathbf{X}' \approx \mathbf{A}\mathbf{X}.$$

Then denoting by $\mathbf{X}^\dagger$ the pseudoinverse of $\mathbf{X}$, the least-squares estimate of $\mathbf{A}$ would be given by

$$\mathbf{A} \approx \mathbf{X}'\mathbf{X}^\dagger. \tag{4.3}$$

The spectral properties of the system are then estimated as usual by the eigendecomposition of $\mathbf{A}$.

For the case of high-dimensional systems, significant computational gains can be realized by reducing the dimensionality of the problem. The rank of $\mathbf{A}$ is limited by the minimum dimension of $\mathbf{X}, \mathbf{X}'$. Instead of studying the spectral properties of the full-state system, we can project the high-dimensional state onto the leading principal components of $\mathbf{X}$ and approximate the spectrum of $\mathbf{A}$ by the spectrum of the matrix that steps this low-dimensional approximation forward in time. That is, if the singular value decomposition of $\mathbf{X}$ is given by

$$X = \mathbf{\Psi}\Sigma\mathbf{V}^*,$$

then the projection of an arbitrary snapshot $\mathbf{x}_k$ onto the leading r principal components is

$$\alpha_k = \mathbf{\Psi}_r^*\mathbf{x}_k,$$

where $\mathbf{\Psi}_r$ consists of the first r columns of $\mathbf{\Psi}$. The spectrum of $\mathbf{A}$ can be approximated by the spectrum of $\tilde{\mathbf{A}}$, where

$$\alpha_{k+1} \approx \tilde{\mathbf{A}}\alpha_k.$$

After some linear algebra, we find that a least-squares estimate for $\tilde{\mathbf{A}}$ is given by

$$\tilde{\mathbf{A}} = \mathbf{\Psi}_r^* \mathbf{X}' \mathbf{V}_r \Sigma_r^{-1} \mathbf{\Psi}_r. \tag{4.4}$$

For this anomaly detection application, the system dimensionality n will typically be much less than the number of available time steps m, so the computation in equation (4.3) is tractable. However, we still employ the dimensionality reduction approach, partly to take advantage of correlations in the time series and partly to make the method scalable to larger problems. To estimate the full system matrix $\mathbf{A}$ would be only a slight modification to equation (4.4):

$$\mathbf{A} = \mathbf{X}' \mathbf{V}_r \Sigma_r^{-1} \mathbf{\Psi}_r.$$

4.1.4 Dynamic Mode Decomposition with Control (DMDc)

The Kalman filter-based anomaly detection method described in Sec. 4.2.1 can be applied to all available sensors simultaneously. The diagonals of the innovation covariance matrix can then be monitored to identify faults in the corresponding sensors. However, each decision tree is trained to output a binary result for a single sensor, although extension to parallel detection is straightforward with multiple decision trees. In this work we therefore restrict our attention to models for a single scalar state x. We can then search for a model as described in section 4.1.2, where related measurements are assumed to be predictors of x. These are treated as "actuation" in the Kalman filter model, but are more accurately taken to be simply exogenous predictors of the sensor measurements.

The full LTI system $(\mathbf{A}, \mathbf{B})$ can now be estimated via a slight modification to the DMD procedure developed by Proctor, *et al.* called Dynamic Mode Decomposition with control (DMDc)[85]. We now split the full state vector into measurements of interest, $\mathbf{x}$, and exogenous "inputs" $\mathbf{y}$. As with the state vectors, the input vectors can be compiled into a single

matrix $\boldsymbol{\Upsilon}$. The estimation then proceeds similarly (as explained in detail in [85]):

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \\ | & | & & | \end{bmatrix}, \quad \mathbf{X}' = \begin{bmatrix} | & | & & | \\ \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_{m+1} \\ | & | & & | \end{bmatrix}, \quad \boldsymbol{\Upsilon} = \begin{bmatrix} | & | & & | \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_m \\ | & | & & | \end{bmatrix}$$

$$\mathbf{X}' \approx \mathbf{A}\mathbf{X} + \mathbf{B}\boldsymbol{\Upsilon}$$

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \end{bmatrix} = \mathbf{X}' \begin{bmatrix} \mathbf{X} \\ \boldsymbol{\Upsilon} \end{bmatrix}^{\dagger}.$$

With the singular value decomposition

$$\begin{bmatrix} \mathbf{X} \\ \boldsymbol{\Upsilon} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Psi}_1 \\ \boldsymbol{\Psi}_2 \end{bmatrix} \Sigma V^*,$$

the matrices $\boldsymbol{\Psi}_1$ and $\boldsymbol{\Psi}_2$ now give the principal components of the state and input subspaces, respectively. Dimensionality reduction based on the singular values Σ is also possible at this stage.

The system $(\mathbf{A}, \mathbf{B})$ is finally estimated by

$$\mathbf{A} = \mathbf{X}' V \Sigma^{-1} \boldsymbol{\Psi}_1^*$$
$$\mathbf{B} = \mathbf{X}' V \Sigma^{-1} \boldsymbol{\Psi}_2^*.$$

4.1.5 Time delays: Hankel-DMD

For complex dynamics, a standard linear system may not have enough descriptive ability to serve as a model for the Kalman filter fault detection method. One approach to enriching the library is to include nonlinear observable functions of the state, leading to "Extended DMD" or "Koopman Mode Decomposition" [129]. EDMD/KMD has been demonstrated to yield models that are predictive enough for accurate full-state estimation with Kalman filters [112], and for sensor fault detection in a power grid model [113]. In the latter work, anomalies were detected using hypothesis testing for the distribution of normalized innovation squared in the Kalman filter under the assumption of Gaussian white noise for process disturbances.

However, accurate EDMD/KMD models rely on a judicious choice of observable functions, which can be challenging in practice; the data matrices quickly become ill-conditioned as the number of observables is increased.

For systems that cannot be accurately represented with standard DMD, we instead augment the library with time-delayed measurements. The use of time-delays in system identification has a long history, including the successful Eigensystem Realization Algorithm and Observer Kalman Identification [50, 51], and deep connections to dynamical systems theory [115, 22]. Augmenting the state vector with time-delays allows the model to capture some of the effects of latent variables. For example, consider a simple harmonic oscillator in periodic sinusoidal motion. A first-order one-dimensional linear model is only capable of expressing exponential growth and decay, not oscillatory dynamics. However, if the state is augmented by a time delay of 1/4 period, a linear model can effectively capture the second-order dynamics (or the latent, imaginary component of motion). Applying DMD to a time-delay-augmented vector can therefore give highly accurate representations of quasiperiodic dynamics [27].

The modeling and estimation procedure is effectively the same, except that a scalar measurement x_k at time t_k is replaced by a vector $\mathbf{x}_k = \begin{bmatrix} x_k & x_{k-d} & x_{k-2d} & \cdots & x_{k-n_d d} \end{bmatrix}^\top$, where d is the length of each delay and n_d is the number of delays. For anomaly detection applications, only the innovation corresponding to the current time step is tracked.

4.1.6 Decision trees

Decision trees are a popular machine learning method for both classification and regression problems. We are interested in classification: for each set of measurements at a given time point we want to determine whether or not a particular sensor has failed. A decision tree is a trainable series of conditional expressions used to arrive at a decision. For example, Figure 4.2 shows a simple decision tree for helping one decide whether or not to pack an umbrella. To answer the question "Do I need an umbrella?" the tree asks whether it is currently raining outside. If so, an umbrella is certainly needed. If not, it asks whether or

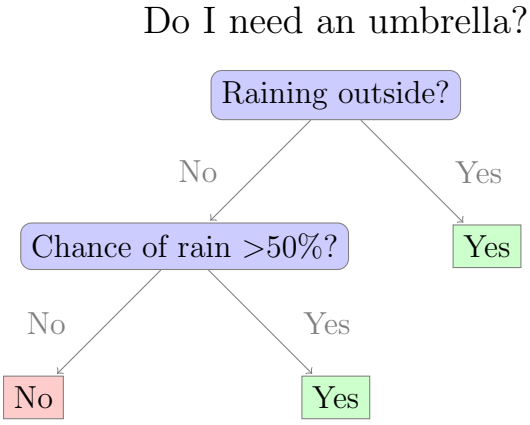


Figure 4.2: A decision tree for deciding whether or not to bring an umbrella when leaving the house.

not the chance of rain exceeds 50%. If so, one should bring an umbrella. If not, the tree suggests leaving the umbrella behind.

Applied to the problem of classifying sensor failure events, the information given to the tree will generally be numeric, so all of its decision rules will resemble the "Chance of rain >50%?" node. Each node in the tree will have associated with it a particular feature and a threshold. When tasked with making a decision about a particular data point, the node will make its choice based on whether the feature value for the data point lies above or below its threshold value. Figure 4.3 shows an example of a decision tree with the same general structure as that of Figure 4.2, but with features relevant to detecting anomalous sensor events. This tree first checks whether the sensor exhibits oscillations (this is replaced by some feature summarizing the oscillatory behavior of the sensor data in the true model), then whether or not the angle of attack is larger than six.

To train a decision tree one must specify a training set

$$\mathcal{D}_{train} = \{(\mathbf{x}_k, y_k) : k = 1, 2, \ldots, N_{train}\},$$

where $(\mathbf{x}_k, y_k) \in \mathcal{D}_{train}$, consists of a feature vector $\mathbf{x}_k \in \mathbb{R}^{N_{feat}}$ and a label $y_k \in \{0, 1\}$. In our case each point corresponds to the sensor measurements at a point in time. Note that

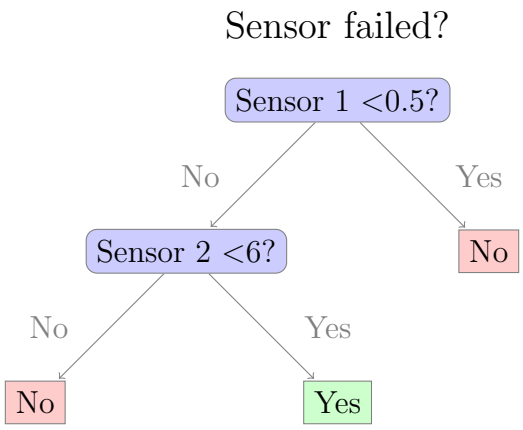


Figure 4.3: A simple decision tree for detecting sensor failure events. Note that the model only flags data points if Sensor 1 has a value less than 0.5 *and* Sensor 2 is large enough.

the data from a single flight test typically yields hundreds of thousands of data points. The entries of $\mathbf{x}_k$ are either direct sensor data or features derived from sensor data (e.g. the median of the previous three measurements). The label y_k is 0 if the sensor is behaving normally and 1 after it has been compromised. The tree is trained with the CART algorithm [17], which recursively adds nodes with corresponding feature thresholds which best split the data points based on their labels. We use the Scikit-learn decision tree implementation [83].

Once a decision tree is trained, it can be used to make predictions on unseen data. If one used custom features to train the model, the same features will also need to be computed for each data point upon which the model is to be used. As we will see in Section 4.3, all of the features we use can be readily computed **online**, that is, in a regime where the algorithm is used to detect anomalies with real-time data and without any additional pretraining.

The *depth* of a decision tree pertains to the maximum number of questions that must be answered before a classification is made. Figures 4.2 and 4.3 both show trees with depths of two.

Deeper trees can better model interactions between features and are better at classifying data from $\mathcal{D}_{train}$. However, excessively deep decision trees are more likely to overfit, generalize poorly, and exhibit poor performance on the validation or test sets. Tree depth is a

hyperparameter that can be tuned with cross-validation.

The number of features, N_{feat}, can impact the performance of the model in two ways. If it is excessively large, the decision tree will take much longer to train and it may become prone to overfitting. If it is too small, or if the features provided are only very weakly correlated with the labels, the decision tree may be unable to learn enough to make accurate predictions. For this application, $N_{train} \gg N_{feat}$, so it is very unlikely that a large number of features would cause the decision tree to overfit the training data.

Pros and cons

Decision trees have many positive qualities, but for this application the following are the most relevant.

Pros:

- Interpretable decision mechanism: to determine how a decision tree is classifying points, one can simply inspect the rules it uses to perform the classifications.

- Extensible: It is trivial to add new features to a decision tree. If a flight test engineer were to tell us that they thought the cube-root of the aircraft velocity should have predictive power, it would be extremely easy to train and test a new decision tree using this extra feature.

- Fast to train and evaluate. Decision trees are extremely compact in terms of the number of parameters needed to specify a given tree. As a result, they take very little time to train, even on vast quantities of training data, and they can be evaluated exceptionally quickly on unseen data.

- Automatically identifies best features: The decision tree implementation we use can, upon being trained, tell the user which features were most useful to it. Since decision trees are so fast to train and evaluate, this allows one to perform rapid iterations of

testing lots of features, identifying the most salient among them, and pruning the least informative.

- No need to scale or normalize features: Decision trees perform just as well with or without scaled and/or normalized data.

Cons:

- Requires good features to perform well: Decision trees are only as good as the features fed to them. Moreover, their decision boundaries are always *orthogonal* to the features given to them. This is because they always make decisions using threshold values. If it turned out that the sum of two measurements was the best predictor of sensor failure, the decision tree would only be able to make use of that information if the user were to create a new feature for the sum.

- Prone to overfitting: Without regularization, a decision tree will tend to overfit the training data. We have chosen to go the route of employing strong regularization by limiting our decision trees to be relatively shallow.

4.2 The proposed method

A high level overview of the method follows. We break the process into an offline phase wherein the model is calibrated using training data, and an online phase where the model is deployed to detect sensor fault events in real time.

1. **Offline phase:** Given a training set $\mathcal{D}_{train}$ consisting of sensor measurements at various time points and corresponding labels,

   (a) Compute the DMDc system $(\mathbf{A}, \mathbf{B})$ as in Section 4.1.4 using time-delayed measurements described in Section 4.1.5;

(b) Derive any desired features from $\mathcal{D}_{train}$ to be used in conjunction with the decision tree. This includes the average innovation covariance (4.2), which can be computed using $\mathbf{A}$ and $\mathbf{B}$;

(c) Train the decision tree using the derived and raw features.

2. **Online phase:** Given a series of measurements $\mathbf{x}_1, \mathbf{x}_2, \ldots$

(a) Compute any features expected by the decision tree, using the previously constructed $\mathbf{A}$ and $\mathbf{B}$ to compute the average innovation covariance;

(b) Pass the features into the decision tree to obtain a class prediction (either that the sensor has failed or continues to function properly).

4.2.1 Computing the average innovation covariance

Combining the previous discussions, the steps necessary for forming the DMDc and Kalman filter and computing the innovation covariance are as follows:

1. Split the time series of available sensor data into measurements of interest $\mathbf{x} \in \mathbb{R}^n$ and (presumably correlated) "inputs" $\mathbf{y} \in \mathbb{R}^p$. The state $\mathbf{x}$ and input $\mathbf{y}$ may include time-delayed values for richer descriptive capability, as detailed in Sec. 4.1.5. We will seek a system that predicts the next measurement of interest as

$$\mathbf{x}_{k+1} \approx \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{y}_k$$

2. Form time-stepped matrices $\mathbf{X}, \mathbf{X}'$, and $\boldsymbol{\Upsilon}$:

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \\ | & | & & | \end{bmatrix}, \quad \mathbf{X}' = \begin{bmatrix} | & | & & | \\ \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_{m+1} \\ | & | & & | \end{bmatrix}, \quad \boldsymbol{\Upsilon} = \begin{bmatrix} | & | & & | \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_m \\ | & | & & | \end{bmatrix}$$

3. Compute the singular value decomposition

$$\begin{bmatrix} \mathbf{X} \\ \boldsymbol{\Upsilon} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Psi}_1 \\ \boldsymbol{\Psi}_2 \end{bmatrix} \Sigma V^*,$$

where $\boldsymbol{\Psi}_1$ contains the first n rows of the full SVD, and $\boldsymbol{\Psi}_2$ contains the remaining p. These represent the principal axes of state space and input space, respectively.

4. Examine the singular value spectrum (diagonal elements of Σ). These represent the relative variance of the data explained by each of the principal components. A sharp drop off in the singular value spectrum indicates large correlations within the measurements or inputs, and rank-truncating the SVD may be helpful to stabilize the system (or reduce the dimensionality, for large systems). This is done to rank r by retaining only the first r columns of $\boldsymbol{\Psi}_1, \boldsymbol{\Psi}_2$, and V, and the first r diagonals of Σ.

5. Estimate the LTI system $(\mathbf{A}, \mathbf{B})$ by

$$\mathbf{A} = \mathbf{X}' V \Sigma^{-1} \boldsymbol{\Psi}_1^*$$
$$\mathbf{B} = \mathbf{X}' V \Sigma^{-1} \boldsymbol{\Psi}_2^*.$$

6. *Online*, maintain a Kalman filtered estimate of the measurements of interest by updating the observer system

$$\hat{\mathbf{x}}_{k+1} = \mathbf{A}\hat{\mathbf{x}}_k + \mathbf{B}\mathbf{y}_k + \mathbf{K}(\mathbf{x}_k - \hat{\mathbf{x}}_k).$$

The Kalman gain $\mathbf{K}$ can either be optimally estimated by solving a Riccati equation, or chosen by hand if the model is simple enough. If the filter is developed as a feature for a decision tree, $\mathbf{K}$ is a scalar and is related to the time sensitivity of the feature.

7. Track a moving average of the innovation covariance

$$V_k = \frac{1}{N} \sum_{i=k-N}^{k} (\mathbf{x}_i - \hat{\mathbf{x}}_i)(\mathbf{x}_i - \hat{\mathbf{x}}_i)^\top.$$

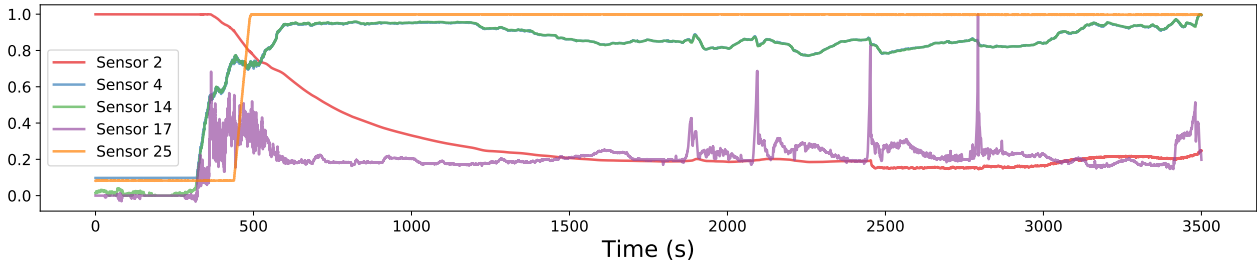


Figure 4.4: A plot of data from a subset of the sensors for test flight 1. Measurements from the faulty sensor (Sensor 2) are shown in red. The sensor fails just before second 2500, where there is a short drop followed by erratic, noisy measurements. We chose which features to plot based on the feature importances returned by a decision tree trained on raw sensor data to predict failure events.

Persistently large values indicate an anomaly. These are detected in an automated fashion by then feeding the innovation covariance to a decision tree as a feature, along with other sensor measurements.

4.3 Three example applications

4.3.1 Real-world dataset

We first consider detecting sensor failure using anonymized (scaled to lie in $[-1, 1]$) measurements from aircraft sensors collected during flight tests. We focus on detecting failure in a given sensor that has a high failure rate (sensor 2) using the data from 25 other on-board sensors. Quite a few of the sensors capture similar or redundant information which the DMD model will exploit. In total there are 21 flights, each roughly seven hours in length, with measurements recorded at a frequency of 20 Hz. The sensor in question fails in 14 of the test flights. There is one time series in which the sensor failure was detected, the sensor was fixed, then the sensor broke again. This case was split into two separate time series. When sensor 2 fails there is typically a small constant shift in the data followed by increased noise for the duration of the flight. Such a failure is shown along with anonymized data from some of the more relevant sensors in Figure 4.4.

With this dataset we are interested in answering the following questions:

1. How soon can the model detect sensor faults? How many examples were misclassified? Were positive or negative examples more likely to be misclassified?

2. Can we distinguish sensor failure events from other events, such as anomalous measurements resulting from special test conditions or maneuvers?

As the data is collected from actual flights the sensor failure time is generally evident, but is not precisely defined. We have hand-labeled estimated time of failure in each case; reported time to detection is based on these estimates. Systematic error in the absolute detection time is therefore possible, but comparisons between flight tests should be reliable.

In order to train the decision tree the flights are divided into a training and testing set (no validation step is used as all parameters are determined with cross-validation using the training data). The training set consists of data from six test flights, flights 0 through 5, four of which contain a sensor failure.

The DMD model is calibrated using flight 3, which contains no failure events. Notably, the model learns to predict future values of Sensor 2 by averaging together the current measurement from Sensor 2 with those of three other sensors with redundant signals (Sensors 0, 1, and 10). Each is given roughly equal weight. This is reminiscent of a weighted-mean method for consolidating redundant measurements into one fault tolerant estimate [4]. From Figure 4.1 it is easy to see that something like the difference between sensors 2 and 10 would be a good proxy for when sensor 2 is behaving properly, but we stress that the model figures this out *automatically*.

A decision tree is then trained to predict sensor faults with its hyperparameters being selected using five-fold cross validation. Examples are reweighted before being fed to the tree in order to mitigate the effects of class imbalance. The top ten signals along with their feature importances (with respect to the decision tree) are given in Table 4.1.

Feature	**Importance**
V_k	0.7514
Sensor 25	0.1985
Sensor 4	0.0261
Sensor 5	0.0095
Sensor 9	0.0057
Sensor 14	0.0056
Sensor 17	0.0012
Sensor 22	0.0008
Sensor 13	0.0005
Sensor 7	0.0002

Table 4.1: Feature importance for the decision tree trained using the flight test dataset.

As seen in Table 4.1 most of the importance is concentrated in just two of the features, with the majority of the weight going to the window-averaged innovation covariance. The second-ranked feature, Sensor 25, turns out to be an almost binary signal indicating when measurements should be collected from the sensor of interest.

The results of the detection of sensor failures on the flights in the test data set is summarized in table 4.2. The results are presented for each individual flight so that poor results can be more easily investigated. As there are so many data points for each flight, standard performance metrics such as precision (the proportion of flagged examples that were actually true positives) and recall (the proportion of true positives we were able to detect) are not particularly helpful here[1]. Instead we focus on the number of false positives and false nega-

[1]True positives and true negatives are positive (failed sensor) and negative (working sensor) examples correctly classified by the model. False positives are negative examples the model classified as belonging to the positive class, i.e. instances where the sensors are working properly, but the model erroneously predicts a sensor has failed. Similarly, false negatives are positive examples the model thought were negative examples.

Flight	Total examples	False positives	False negatives	Accuracy	Lag Time (s)
6	175,162	708	0	0.995958	N/A
7	213,964	0	22	0.999897	1.1
8	143,140	0	4,663	0.967424	10.1
9	121,413	0	412	0.996607	1.35
10	368,140	0	0	1.000000	N/A
11	152,146	0	130	0.999146	6.5
12	278,465	50	0	0.999820	N/A
13	372,950	3	0	0.999992	N/A
14	124,570	2,617	0	0.978992	N/A
15	166,880	0	0	1.000000	0
16	302,550	0	11,690	0.961362	∞
17	295,680	0	63	0.999787	2.9
18	64,700	0	9	0.999861	0.45
19	472,090	5,9809	157	0.872978	7.85
20	117,650	0	229	0.998054	11.45

Table 4.2: Prediction results on test flights

tives along with lag time (time from actual sensor failure to detection). Overall accuracy is also included to give more context to the number of false positives and negatives reported.

With the exception of two flights, 16 and 19, the results are promising if the goal is to alert a human of a potential sensor failure during a flight test within a few seconds of occurrence. There is some fluctuation in lag time, most likely due to circumstances that affect flight dynamics. The false negatives are almost entirely due to lag time. The false positives tend to persist briefly (about two seconds) before automatically correcting. For example for flight 8 we observe four different short periods when the model thinks a sensor

failure has occurred. These correspond to abrupt changes in flight conditions as the pilot(s) carry out different test maneuvers.

Now we turn our attention to the two flights with poor results. In flight 16 there are a large number of false negatives (and a lag time of ∞) as the model completely misses an actual sensor failure. This is because sensor 25, which is normally active during the tests, was completely inactive during this flight. If one wished to learn to predict regardless of whether or not this sensor was in use, one may need to resort to training a separate model using only data where the sensor was inactive because it is used in all the other flights.

In flight 19 we see a large number of false positives. This is caused by persistent discrepancies between sensor 2 and its redundant sibling, sensor 10. We suspect that sensor 10 temporarily malfunctioned. Whether or not it is desirable for the algorithm to flag these anomalous measurements depends on the application. After the abnormal behavior of sensor 10 the model detects the failure of sensor 2 almost perfectly.

The final model is able to reliably detect true sensor failure events within 12 seconds (with most occurring in under five). There are infrequent false positives detected with a brief persistence of about two seconds. In some cases the measurements taken by the faulty sensor drift close to the values of the reliable redundant sensors, leading to false negatives.

4.3.2 Synthetic datasets

In order to better understand the capabilities of our proposed approach we apply it to two sensor failure tasks derived from simulated datasets. In both instances we use a dynamical model to generate realistic sensor data, which we then augment in various ways to mimic common sensor failure modes. Specifically, we simulate the faults shown in Figure 4.5, similar to those studied by Eykeren and Chu [38]: catastrophic failure (multiplicative), slow oscillation (additive), increased noise (additive), and slow drift (additive). In every case we add a small amount of white noise (mean 0, standard deviation 5×10^{-3}) to the underlying sensor measurement of interest *after* incorporating the fault.

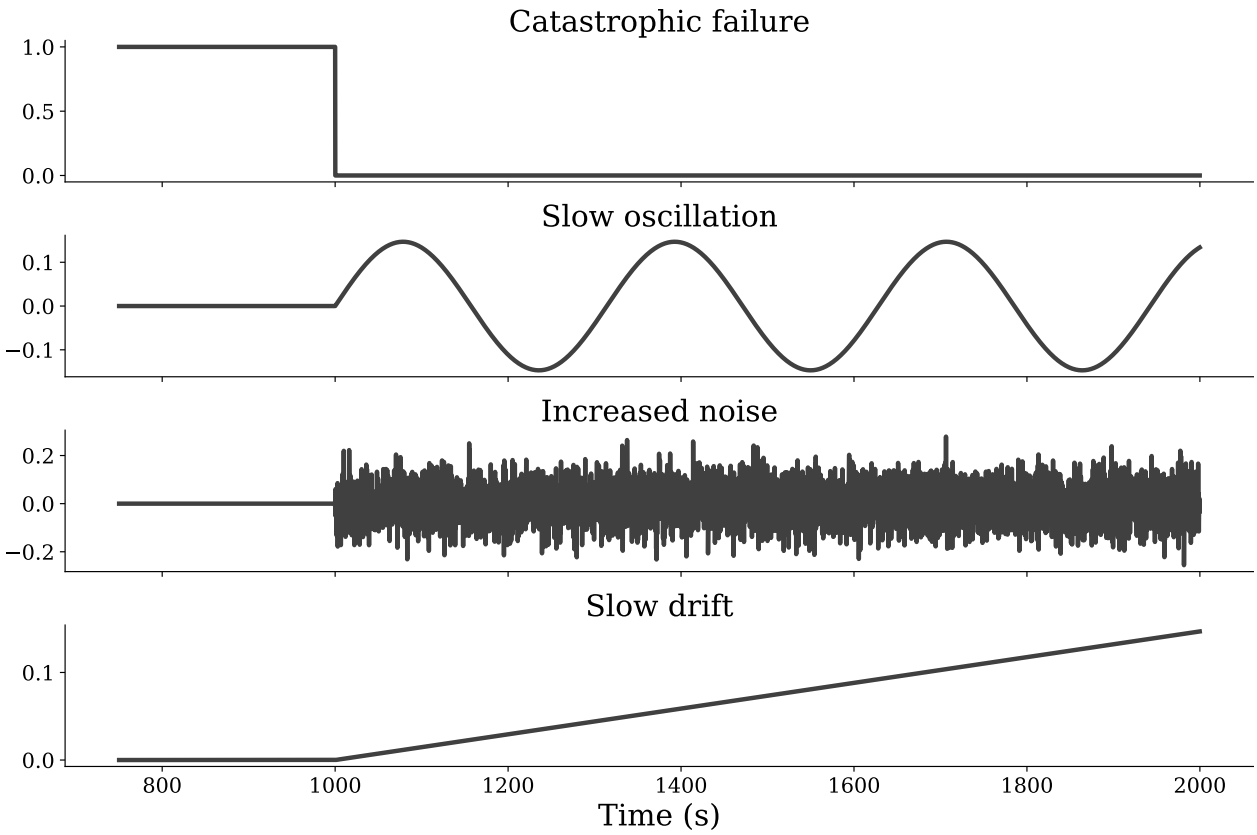


Figure 4.5: The sensor faults applied to the simulated datasets. In particular, the faults used for the dataset of Section 4.3.2 are shown here. Those used for the dataset of Section 4.3.2 are the same up to rescaling.

Goman-Khrabrov model

A long-standing challenge in aerodynamic modeling was capturing the effect of a separated flow on aerodynamic moments. This arises for example in high angle-of-attack maneuvers, when an airfoil wake can detach, leading to much more complex physical behavior such as dynamic stall [59]. The aerodynamic moments in this case depend not only on the airfoil configuration, as in standard linearized approaches based on stability and control derivatives, but also on the state of the separated flow.

Goman and Khrabrov proposed a mathematical model that treats the flow state as a dynamic internal system variable [43]. For the case of a high angle-of-attack airfoil this is a scalar variable $x \in (0, 1)$ representing the separation point normalized by the chord length, so that fully attached flow corresponds to $x = 1$. The internal flow field dynamics are modeled with a simple time-delay model:

$$\tau_2 \dot{x} + x = x_0 \left(\alpha - \tau_1 \dot{\alpha}\right).$$

The function $x_0(\alpha)$ defines the empirical steady separation point as a function of angle of attack α. Quasisteady effects are expressed through the time-delay shift $\tau_1 \dot{\alpha}$ The overall model dynamics are a relaxation towards the quasisteady separation point on a timescale τ_2. The moments are then algebraic functions of the aerodynamic state, for example $C_L = C_L(\alpha, x)$. These are derived with standard modeling methods. For a high angle-of-attack airfoil,

$$C_L(\alpha, x) = \frac{\pi}{2} \sin\left[\alpha(1 + \sqrt{x})^2\right].$$

This model was shown to accurately describe experimental data for a NACA 0015 airfoil [43].

For our synthetic data, we use a simple model for the steady separation point:

$$x_0(\alpha) = \frac{1 - \tanh\left[20(\alpha - 0.25)\right]}{2}.$$

The time constants τ_1 and τ_2 in the model can be obtained in general by fitting to experimental data. We use the reported values $\tau_1 \approx 0.5$ and $\tau_2 \approx 4.5$, nondimensionalized by chord

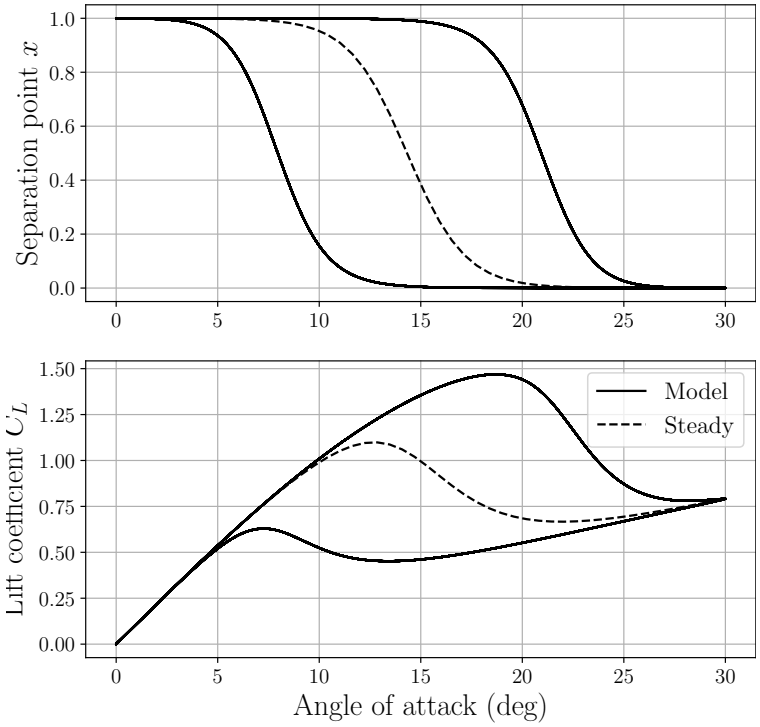


Figure 4.6: Hysteresis in flow separation (top) and lift coefficient (bottom) in the Goman-Khrabrov model for an airfoil undergoing sinusoidal pitching motion at nondimensional frequency $\omega = 0.05$. Stall is delayed relative to the steady value for pitch-up motions with attached flow (upper curves on both plots).

length and free stream velocity.

The separation point and lift coefficient for a sinusoidal pitching motion at nondimensional frequency $\omega = 0.05$ is shown in figure 4.6, along with the steady values as a function of angle of attack. The effect of the model is to capture observed hysteresis in the curves; the flow remains attached to higher angle-of-attack on pitch-up motions and stall is delayed. Conversely, when the flow is separated during a pitch-down maneuver it remains so for longer, resulting in reduced lift relative to the steady value.

Our experimental dataset consists of measurements of C_L, α, and $\dot{\alpha}$ taken five times per second for 2000 seconds. The DMDc model is trained using the full time-series. We then simulate failure of a hypothetical C_L sensor at $t = 1000$ using each of the fault modes shown

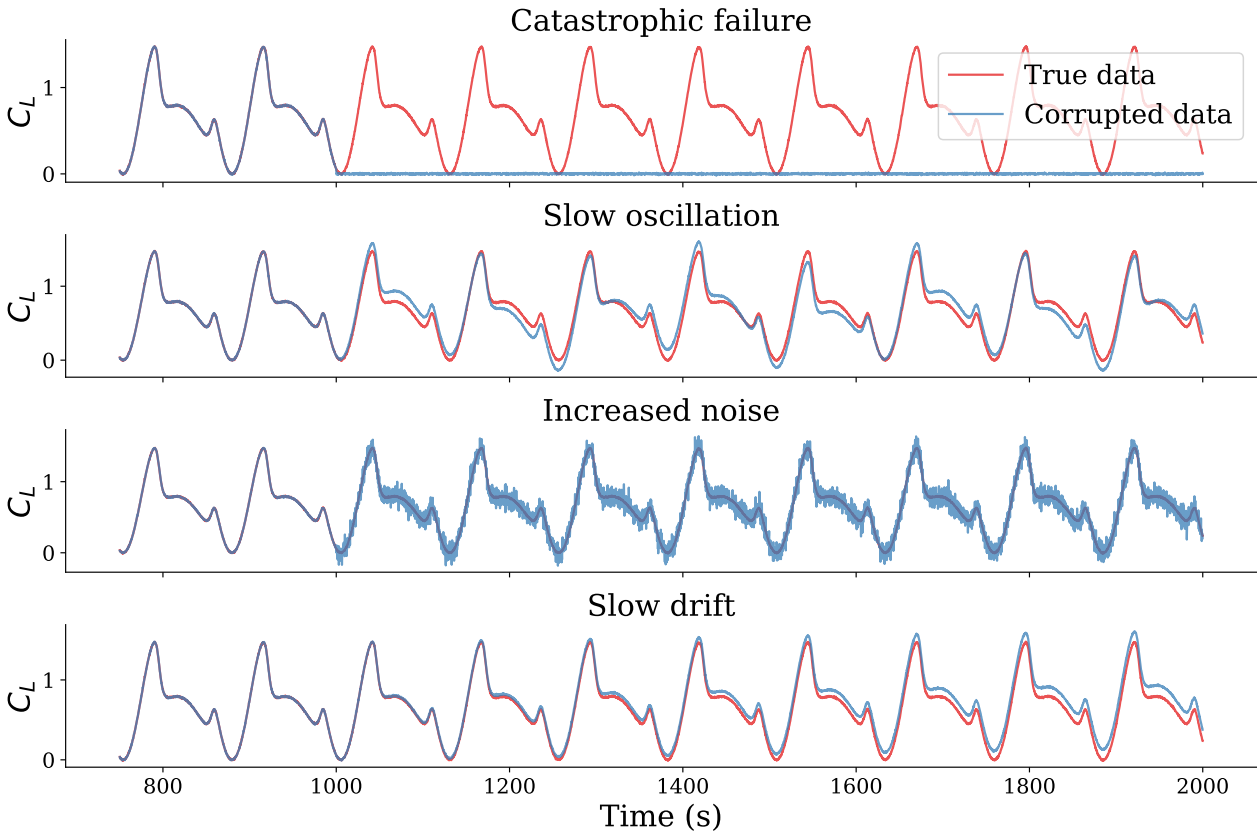


Figure 4.7: Visualizations of the different lift coefficient (C_L) sensor failure modes for the Goman-Khrabrov model. Sensor failure occurs at $t = 1000$. Note that we omit from this plot measurements taken with $t < 750$.

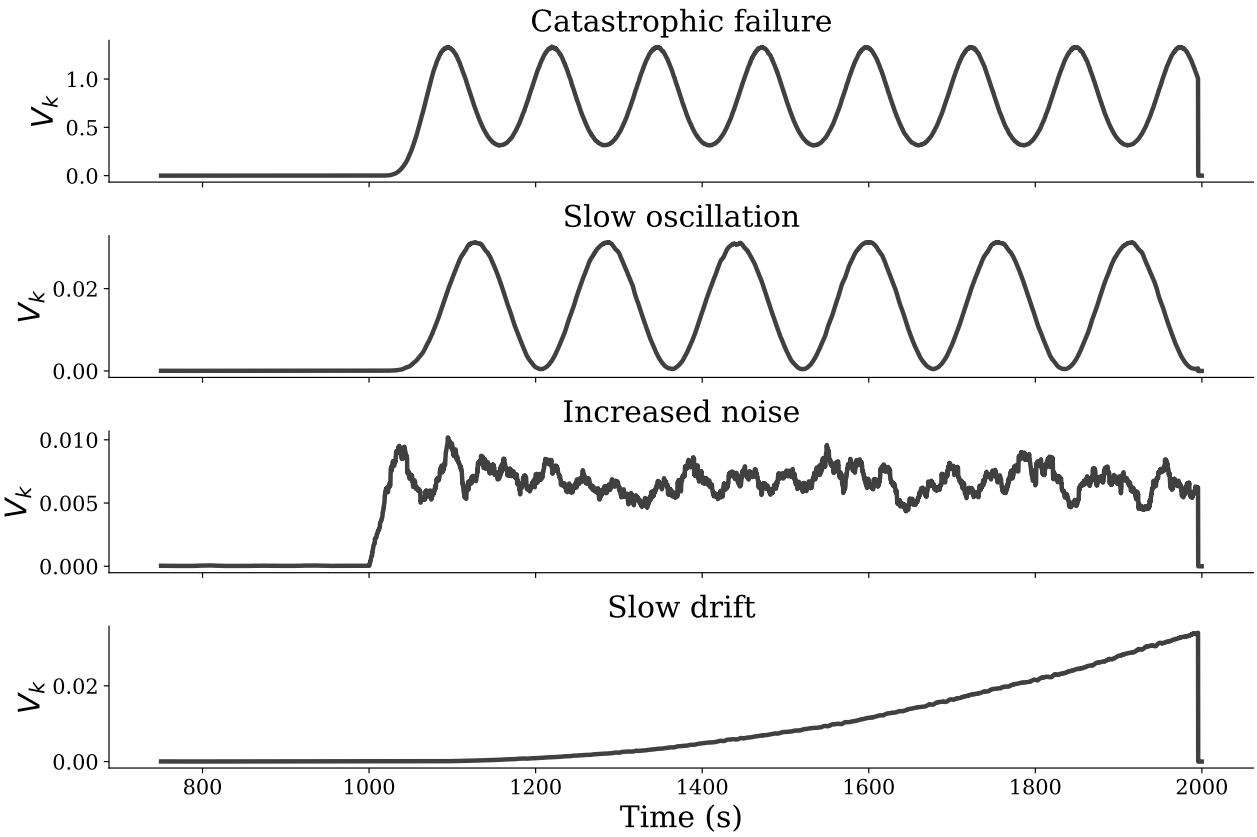


Figure 4.8: Moving average of the innovation covariance for each sensor fault type using data generated with the Goman-Khrabrov model. Sensor failure occurs at $t = 1000$.

in Figure 4.5, which results in the time series given in Figure 4.7. We then precompute the innovation covariance, V_k, for each sensor failure type, with the Kalman filter (4.1) constructed using DMDc. Note that the scale of V_k differers between the modes so only training on one type of sensor failure may lead to poor results when trying to detect different failure types. We plot the moving average of the covariance for the different failure modes in Figure 4.8. Note the differences in scale between the modes. A threshold for V_k, above which a sensor failure is deemed to have occurred, which is chosen based on only one type of fault, may be inappropriate for the others.

Finally, we train a decision tree to predict when the a sensor failure has occurred. The tree is given access to α, $\dot{\alpha}$, the corrupted C_L measurements, and V_k as features. At each

Model	Fault types seen	Acc.	Precision	Recall	In-group acc.	Depth
1	Catastrophic failure	0.9732	0.9980	0.9483	0.9343	5
2	Slow oscillation	0.9672	0.9629	0.9718	0.9351	**2**
3	Increased noise	0.9730	0.9993	0.9467	0.9482	**2**
4	Slow drift	0.9784	0.9707	**0.9867**	0.8678	3
5	All	**0.9818**	**0.9996**	0.9642	**0.9823**	5

Table 4.3: Performance metrics for models trained on different subsets of sensor fault types with data generated with the Golman-Khrabrov model. "Depth" refers to the depth of the decision tree chosen during cross-validation. The best values for each column are shown in bold.

time point the model must attempt to predict whether the given C_L measurement has been corrupted or not (whether the sensor has failed). We use five-fold cross-validation to select model parameters. We construct two types of training and testing data; the first involves incorporating random examples from *all four* failure types into the training set and the second sources its training data from just one sensor fault and attempts to predict when the others have occurred. Note that all training and test sets were constructed to be of equal size (in every case the training set consists of 30000 examples and the testing set of 10000 examples).

Model performance is summarized in Table 4.3. "In-group accuracy" refers to the accuracy of the model on a holdout set during cross-validation. For models 1-4 this number gives an estimate of the models' accuracies on the same fault type on which they were trained. The best accuracy and precision scores are both achieved by the model with training data from all four fault types. However, model 4 has the best recall, meaning that it misses the fewest sensor failure events. This is likely due to the fact that model 4 must choose a very small threshold for V_k at which to separate the negative and positive class instances. Choosing a threshold of 0 would result in a recall of 1, but a precision of 0.5 as all examples would be classified as sensor failures.

The proposed method is able to reliably detect sensor faults for data generated from the Goman-Khrabrov model, even on unseen fault types. However, we note that if either the amplitude or frequency of the forcing term changes after the DMD model has already been trained, the DMD model becomes too inaccurate to be useful and the predictive performance of the overall model suffers considerably. This is a general drawback of data-driven models: when training and testing sets are different enough in distribution, models learned on one set have a hard time generalizing to the other.

Flight dynamics model

Motivated by anomaly detection in a flight test setting, we also consider a longitudinal flight dynamics model for a business jet in atmospheric turbulence [110]. The equations of motion for the longitudinal model capture motions in the forward and vertical directions, including pitch for a total of three degrees of freedom. Flight controls are included for elevator, thrust, flaps, and stabilator; these are trimmed for steady, level flight. The aerodynamic model includes a realistic geometric configuration, stability and control derivatives, US standard atmospheric conditions interpolation, and a Mach number correction. The dynamics are forced by atmospheric turbulence generated to approximate the von Kàrmàn spectrum by filtering band-limited white noise [73, 126].

From this system we can measure not only the dynamic variables for inertial velocity and pitch, but also lift, drag, pitching moment, true airspeed, angle of attack, and Mach number. We consider measurements of the true airspeed (TAS), informed also by angle of attack, inertial airspeed, pitch, lift, and thrust. Note that the flight controls are constant, but the model includes an altitude correction for effective thrust. We generate samples for each variable over $t \in [0, 600]$ at a rate of 10 samples per second. Next we build a Kalman filter from a DMDc model trained on the TAS data. Various faults are then introduced starting at $t = 300$ in the TAS sensor to obtain the time series shown in Figure 4.9. Note that these measurements are much noisier than those from the previous section due to the turbulence-based forcing.

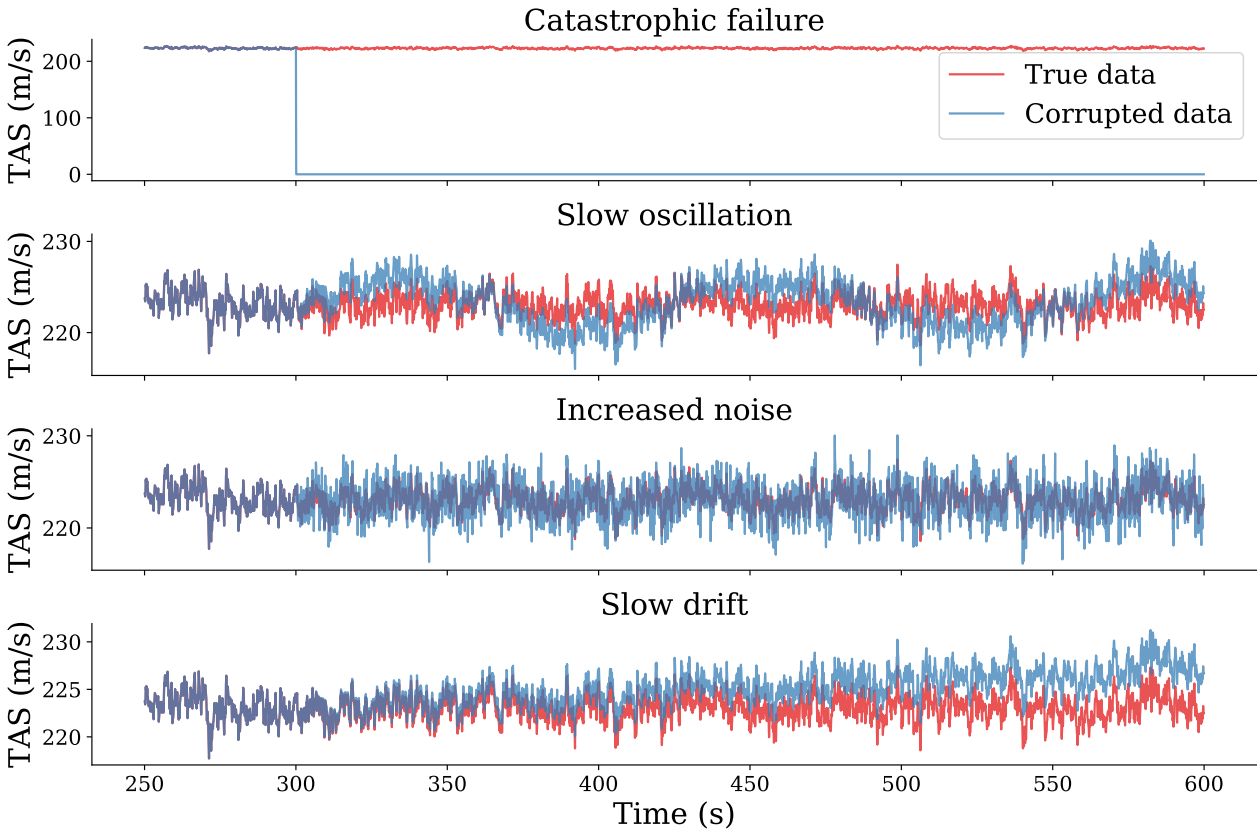


Figure 4.9: Visualizations of the different lift coefficient (C_L) sensor failure modes for the longitudinal flight model. Sensor failure occurs at $t = 300$. Note that we omit from this plot measurements taken with $t < 250$.

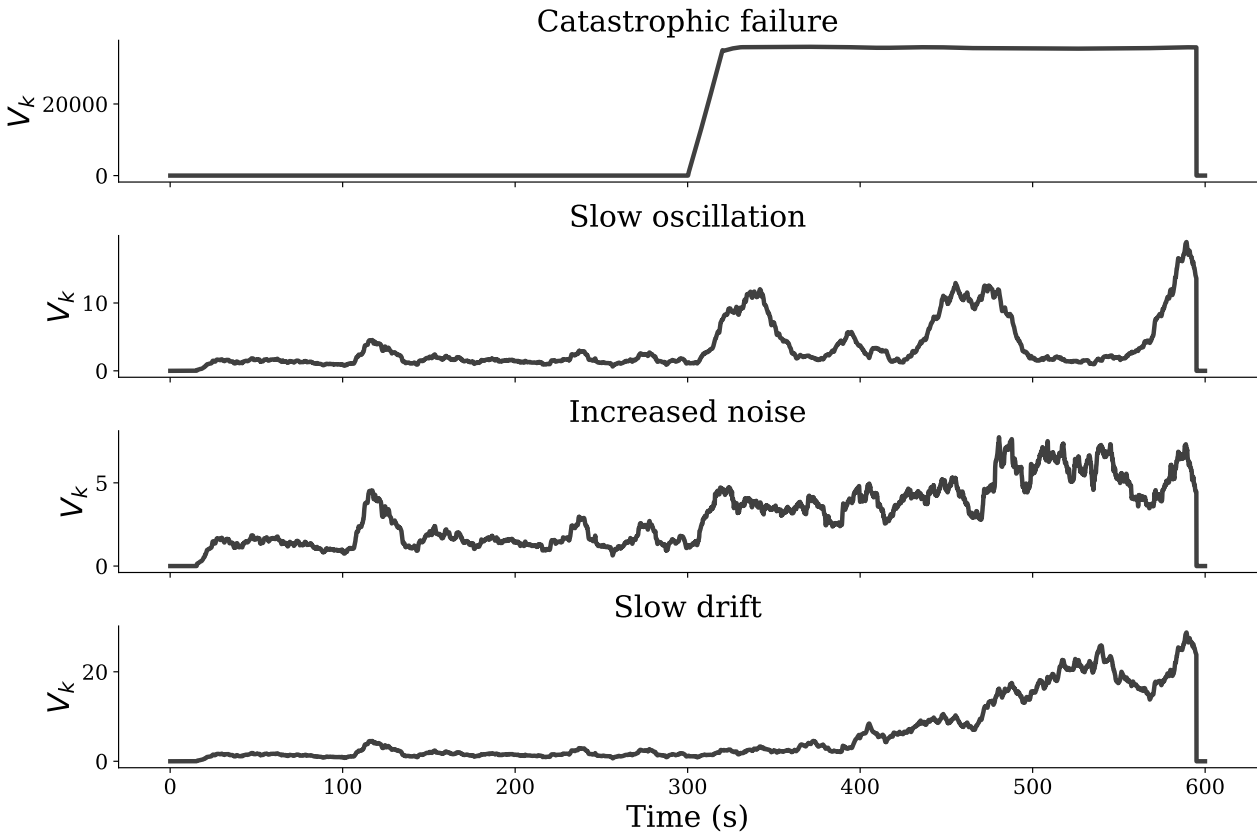


Figure 4.10: Moving average of the innovation covariance for each sensor fault type using data generated from the the flight dynamics model. Sensor failure occurs at $t = 300$.

Again we precompute the innovation covariance, V_k, for each fault type. The results are shown in Figure 4.10. It is evident from this plot that all faults but the catastrophic failure will be difficult to detect. None of the other covariance time series admit a single threshold that distinguishes bad readings from good ones. As before we study the behavior of decision trees trained on a data from all four failure modes or from just one mode. The trees are given measurements of TAS, angle of attack, inertial air speed, pitch, lift, thrust, and V_k. Training and testing sets are of equal size, with each training set constituting 75% of the data. Since the dataset is more complex we allow our cross-validation procedure to select trees of depth up to seven.

Table 4.4 details the results of these experiments. Model 5 outperforms all the others by

Model	Fault types seen	Acc.	Precision	Recall	In-group acc.	Depth
1	Catastrophic failure	0.5002	0	0	1.0	**2**
2	Slow oscillation	0.9549	0.9230	**0.9926**	0.8412	3
3	Increased noise	0.8974	0.9915	0.8016	0.8878	**2**
4	Slow drift	0.9572	0.9659	0.9478	0.8190	4
5	All	**0.9867**	**0.9961**	0.9767	**0.9859**	7

Table 4.4: Performance metrics for models trained on different subsets of sensor fault types with data generated with the flight dynamics model. "Depth" refers to the depth of the decision tree chosen during cross-validation. The best values for each column are shown in bold.

a considerable margin, though it is also the most complicated of the models tested. Notably, model 1 adopts a classification rule allowing it to predict catastrophic sensor failures with perfect accuracy (its in-group accuracy is 1), but for all other failure types it is no better than a coin toss. The threshold it selects for V_k is much too large for the other fault types and so it almost always predicts that no sensor failure has occurred. In this case model 2 has the highest recall, but also a low precision, for the same reason as model 4 did previously: its threshold for V_k is set lower than the others, reducing the number of false negatives at the cost of more false positives.

The effectiveness of our procedure at identifying sensor faults in simulated flight test data is satisfactory, but there is still a noticeable degradation in performance relative to data simulated with the Goman-Khrabrov model. This is due in part to the complexity of the simulated dynamics as well as the erratic atmospheric forcing. We observe that for this more challenging dataset, it is increasingly important that the model be trained using examples from multiple different types of failure modes. Should this prove to be impossible (e.g. if data for some failure types is unobtainable), then the model should be trained using fault types that are most difficult to detect. Failure modes which can be detected trivially, such as catastrophic sensor failure, may prove insufficient to train a robust detector.

4.4 Conclusions

An automatic algorithm for sensor fault detection in systems with various types of sensor failures was presented. The method first uses the dynamic mode decomposition for control with time-delay measurements to learn a simple linear time-invariant model for the evolution of a sensor of interest in time. This model is embedded in a Kalman observer which is then used to predict future measurements. A potential sensor fault is detected when the predicted and measured sensor values disagree by too large a margin, with the ruling ultimately being made by a decision tree. Each component of the proposed method can be trained in an automated fashion. The performance of the proposed method was demonstrated on three test datasets; one consisting of measurements from a series of flight tests and and two of simulated data from the Goman-Khrabrov and a realistic flight dynamics model. In each case the difference between the true and Kalman-observer-predicted values of the sensor of interest provided an accurate proxy for when sensor failure had occurred.

There are numerous extensions that could be explored for improving upon the results obtained here. Any of the components of the algorithm could be replaced with more sophisticated variants. For example, advances in Koopman theory could be leveraged to enrich the linear time invariant physics model. A nonlinear model such as an extended Kalman filter or a model learned via some other model discovery framework [24, 104] could be used in place of the Kalman filter. Such generalizations would allow for the application of the proposed method to systems exhibiting strongly nonlinear dynamics. The performance of the decision tree could be enhanced by employing an ensemble [61], a cost-sensitive training algorithm [65], or by better utilizing class probabilities output by the tree.

Chapter 5

PORT APPROXIMATION FOR PARAMETRIZED COMPONENT-BASED STATIC CONDENSATION: INTERSECTING PORTS IN 2D

This chapter is based on joint work with Ulrich Hetmaniuk.

5.1 Introduction

Structures built from a set of similar components arise naturally in a variety of engineering and design applications. For instance, buildings may consist of many nearly identical beams, walls, and studs. Furthermore, the physical properties of these components can often be represented as parameters. Specialized computational tools have been developed for the efficient numerical simulation PDEs modeling such component-based structures. The way many of these methods take advantage of the component-based architecture is by performing some form of dimension reduction once for a representative from each component class, then re-using the resulting models for all similar components present in the structure. An early technique for doing so is component mode synthesis, which focuses on capturing the degrees of freedom on the interface between components using an eigenmode expansion [15]. For parametrized PDEs one popular approach is the static condensation reduced basis element (SCRBE) method, introduced in [37]. This algorithm employs reduced basis approximations in the interiors of the components to handle the parameter dependence, while also making use of a static condensation procedure to reduce the degrees of freedom of the system to those on the interface between components. Finally, it reduces the size of the problem further with a port approximation step. The SCRBE method admits a decomposition into an expensive offline phase, and a rapid online phase.

Multiple generalizations of SCRBE have been proposed. An extension of the method to complex parametrized Helmholtz equations is presented in [48], wherein Huynh et al. model acoustic problems in mufflers and horns. In [122], Vallaghé shows how to apply the SCRBE method to a class of parabolic problems. A procedure for solving symmetric eigenvalue problems using the SCRBE method is given in [123]. A generalization of the method to domains with a checkerboard pattern of components, similar to the one presented here, is given in [7].

Recently, work has been done to improve the original SCRBE method. Smetana and Patera describe the port approximation spaces that are optimal in the sense of Kolmogorov in [107]. They also detail a way of computing them. [25] offer a cheaper method for obtaining a close approximation to the optimal modes of [107] using techniques from randomized numerical linear algebra.

The remainder of this chapter is organized as follows. In Section 5.2, we introduce the problem to be solved, notational conventions, and assumptions we make. Section 5.3 presents the main ideas behind our method: port reduction, the static condensation reduced basis element method, and our modifications to it. The section culminates with a summary of the offline-online decomposition for our method. Finally, we describe our numerical experiments and their outcomes in Section 5.4.

5.2 Notation and assumptions

In this section we lay notational and theoretical groundwork for the work presented in this Chapter.

5.2.1 Problem statement

Let Ω be a two- or three-dimensional domain with Lipschitz boundary $\partial\Omega$. Let Γ_D and Γ_N be disjoint parts of the boundary such that $\partial\Omega = \Gamma_D \cup \Gamma_N$.

The problem considered in this work is a second-order differential equation in weak form with homogeneous Dirichlet boundary conditions on Γ_D and Neumann boundary conditions

on Γ_N. The associated bilinear form is

$$a(v,w) = \int_\Omega p\left(\boldsymbol{x}\right) \nabla v\left(\boldsymbol{x}\right) \cdot \nabla w\left(\boldsymbol{x}\right) \ \mathrm{d}\boldsymbol{x}, \tag{5.1}$$

for all functions v and w in the Hilbert space

$$X = \left\{ v \in H^1\left(\Omega\right) : \ v = 0 \ \text{ on } \Gamma_D \right\}.$$

We suppose that $a(\cdot,\cdot)$ is coercive,

$$\exists \alpha > 0, \ \alpha \left(\int_\Omega \left(v^2 + \left|\nabla v\right|^2 \right) \right) \leq a(v,v), \quad \forall v \in X$$

and continuous,

$$\exists \gamma > 0, \ a(v,w) \leq \gamma \left\| v \right\|_{H^1} \left\| w \right\|_{H^1} \quad \forall v, \ w \in X$$

The Lax-Milgram theorem [18, p. 62] states under these conditions that the variational problem

$$\text{Find } u \in X, \ a(u,v) = \int_{\Gamma_N} gv = f(v), \quad \forall v \in X$$

admits a unique solution $u \in X$.

We shall assume that the functions p and g can be parametrized in terms of a finite number of parameters, $\mu_1, \ \mu_2, \cdots, \mu_P \ \in \mathbb{R}$, collected into a vector $\boldsymbol{\mu} \in \mathbb{R}^P$. This assumption enables us to write $a(\cdot,\cdot;\boldsymbol{\mu})$ and $f(\cdot;\boldsymbol{\mu})$ in place of (2) and (3), respectively. The parameters $\boldsymbol{\mu}$ are assumed to lie in a bounded set $\mathcal{D} \subset \mathbb{R}^P$, determined for example by physical or practical considerations. Denoting $a(\cdot,\cdot;\boldsymbol{\mu})$ and $f(\cdot;\boldsymbol{\mu})$[1], there exists a unique solution, $u(\boldsymbol{\mu}) \in X$, satisfying

$$a(u(\boldsymbol{\mu}), v; \boldsymbol{\mu}) = f(v;\boldsymbol{\mu}), \quad \forall v \in X.$$

For the sake of computational efficiency we impose a further assumption on p and g.

---

[1]The linear form f corresponds to integration along Γ_N, for the sake of description. Our work applies to any continuous linear function, defined on X.

Assumption 1. p and g are such that the bilinear and linear forms can be expressed as

$$a(v, w; \boldsymbol{\mu}) = \sum_{q=1}^{Q_a} \theta_a^q(\boldsymbol{\mu}) a_q(w, v)$$

and

$$f(v; \boldsymbol{\mu}) = \sum_{q=1}^{Q_f} \theta_f^q(\boldsymbol{\mu}) f_q(v),$$

where $a_q(\cdot, \cdot)$ and $f_q(\cdot)$ are bilinear and linear forms independent of $\boldsymbol{\mu}$, and $\theta_a^q(\boldsymbol{\mu})$ and $\theta_f^q(\boldsymbol{\mu})$ are scalars which depend on $\boldsymbol{\mu}$.

5.2.2 Domain decomposition

Our method relies on the partition of Ω into disjoint components. For the remainder of this section, we assume that Ω is two-dimensional.

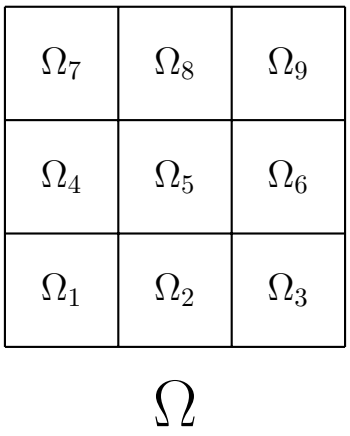


Figure 5.1: The checkerboard domain, Ω

Consider the checkerboard style domain[2], $\Omega \subset \mathbb{R}^2$, shown in Figure 5.1 and decomposed into a set of nine components, Ω_i, satisfying

$$\overline{\Omega} = \cup_{i=1}^{9} \overline{\Omega_i} \quad \text{and} \quad \Omega_i \cap \Omega_j = \emptyset \quad \text{if } i \neq j.$$

[2]Note that, while we have chosen for the sake of convenience to work with a square domain and square components, one could, in principle, apply the method described in this chapter to polygonal or polyhedral domains.

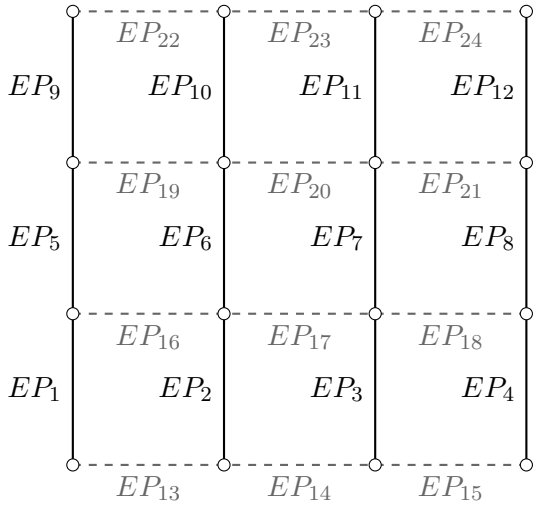


Figure 5.2: Labeling of the edge ports of Ω

We refer to the subdomains, Ω_i, as *components* and to the subsets of the interfaces between components as *ports*, following the nomenclature of Eftang and Patera [37]. We distinguish between two different types of ports: *edge ports* and *corner ports*. An edge port consists of the intersection between the closures of two adjacent components, minus the endpoints. The domain Ω has a total of 24 edge ports which we label $EP_1,\ EP_2, \ldots,\ EP_{24}$. Figure 5.2 shows the orientation of each edge port. An edge port may be adjacent to one or two components.

A corner port occurs at a single point wherever two edge ports meet. This means that a corner port can be adjacent to one, two, or four components simultaneously. The locations of the 16 corner ports for Ω, $CP_1,\ CP_2, \ldots,\ CP_{16}$, are shown in Figure 5.3. Note that for any two adjacent components, Ω_i and Ω_j, the intersection between the closure of the two components is the disjoint union of two corner ports and one edge port. For example, $\overline{\Omega}_5 \cap \overline{\Omega}_6 = EP_7 \cup CP_7 \cup CP_{11}$. We shall sometimes refer to the union of all ports as the *interface* Σ.

Before discussing the discretized formulation, we assume one additional property for the coefficient function, p, namely:

Assumption 2. The function p is generated by a function p_0 defined on $[0,1]^2$, i.e. $\forall i$,

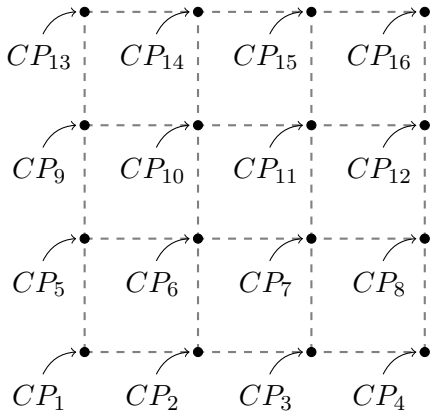


Figure 5.3: Labeling of the corner points of Ω

$\forall \boldsymbol{x} \in \Omega_i$,

$\forall \boldsymbol{\mu} \in \mathcal{D}$,

$$p\left(\boldsymbol{x}; \boldsymbol{\mu}\right) = p_0\left(\xi, \eta; \tilde{\boldsymbol{\mu}}\right) \quad \text{for } (\xi, \eta) \in [0,1]^2 \text{ and } \tilde{\boldsymbol{\mu}} \in \mathcal{D}$$

5.2.3 Discrete finite element formulation

We introduce a piecewise linear conforming finite element space, $X^h \subset X$, of (typically large) dimension, $\mathcal{N}$. The finite element approximation, $u^h(\boldsymbol{\mu}) \in X^h$, to $u(\boldsymbol{\mu})$ is the solution to

$$a\left(u^h(\boldsymbol{\mu}), v; \boldsymbol{\mu}\right) = f(v; \boldsymbol{\mu}), \quad \forall v \in X^h. \tag{5.2}$$

Assumption 3. The finite element space X^h is rich enough to ensure a given level of accuracy for any $\boldsymbol{\mu} \in \mathcal{D}$. That is to say, for any $\epsilon > 0$ and any $\boldsymbol{\mu} \in \mathcal{D}$, we can choose h small enough so that

$$\sqrt{a(u(\boldsymbol{\mu}) - u^h(\boldsymbol{\mu}), u(\boldsymbol{\mu}) - u^h(\boldsymbol{\mu}); \boldsymbol{\mu})} \leq \epsilon.$$

We assume that the mesh associated with the finite element space, X^h, conforms with the components of Ω. To simplify matters, but without loss of generality, we use a mesh based on rectangular elements.

5.3 Approximation method

Here we describe the proposed method. Our algorithm extends the works of Eftang and Patera [37] to the case where *corner ports* are present. It differs from the work of Bader et al. [7] in that it employs a different set of *edge ports*.

A key result, driving the discretization method, is

$$X^h = \left[\bigoplus_{i=1}^{9} \left(X^h \cap \tilde{H}_0^1\left(\Omega_i\right)\right)\right] \oplus V^c = V \oplus V^c$$

where functions in $\tilde{H}_0^1\left(\Omega_i\right)$ are understood to be trivially extended from Ω_i to Ω. Note that functions in V vanish on the ports and that V^c denotes the algebraic complement to V.

The objective is to build a subspace, $X_R^h \subset X^h$ consistent with the decomposition (5.3), such that the discrete solution $u_R^h(\boldsymbol{\mu}) \in X_R^h$, satisfying

$$a\left(u_R^h(\boldsymbol{\mu}), v_R^h; \boldsymbol{\mu}\right) = f\left(v_R^h; \boldsymbol{\mu}\right), \quad \forall v_R^h \in X_R^h,$$

closely approximates the finite element solution $u^h(\boldsymbol{\mu})$ for any parameter $\boldsymbol{\mu} \in \mathcal{D}$.

5.3.1 Port reduction

To reduce the degrees of freedom on the interface, we shall construct a basis for edge port EP_j wherein the $n_e \ll N_e$ functions can replicate "most" of the behavior exhibited by $u^h(\boldsymbol{\mu})$ along EP_j for "most" values $\boldsymbol{\mu} \in \mathcal{D}$.

The basis is generated by an offline training procedure inspired by the one used in [37]. Our training domain, $T = (0,2) \times (0,2)$, consists of four components which can each be identified with $(0,1) \times (0,1)$, see Figure 5.4. On T, we solve

$$\text{Find } z \in H^1(T), \int_T p\left(\boldsymbol{x}; \boldsymbol{\mu}\right) \nabla z(\boldsymbol{x}) \cdot \nabla w(\boldsymbol{x}) d\boldsymbol{x} = 0 \quad \forall w \in H_0^1(T)$$

where z satisfies *inhomogeneous* Dirichlet conditions on ∂T, for a *large* number of random values of $\boldsymbol{\mu} \in \mathcal{D}$ and for a *large* number of arbitrary Dirichlet conditions. Note that we can train on T without any loss of generality, thanks to Assumption 2.

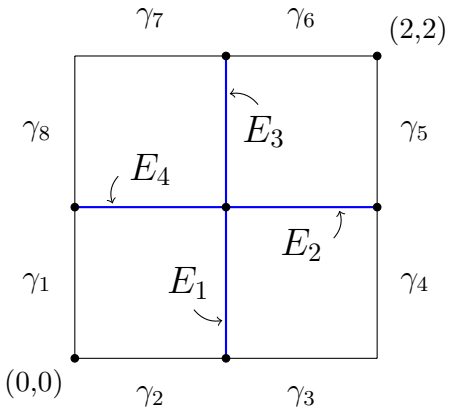


Figure 5.4: The domain, T, used in our training procedure

Next we extract the trace of each solution along the ports separating components (the blue lines, E_1, E_2, E_3, and E_4) and remove the values of each snapshot at the two endpoints by subtracting a linear function. The result is a set of modified one-dimensional snapshots which are zero at both endpoints. The proper orthogonal decomposition (POD) is applied to these modified snapshots to produce a set of orthonormal functions which are zero at both endpoints. We refer to these functions as *port modes*, $\{\chi_k\}_{k=1}^{N_e}$.

For each edge port, EP_j, we first interpolate χ_k on the edge, denoted as $\mathcal{I}_j(\chi_k)$. Then we lift the interpolant onto the interior of the adjacent components via harmonic extension. These extensions are denoted $\Psi_{EP_j,k}$ and are supported on, at most, two components. If Ω_i is adjacent to EP_j then the restriction of $\Psi_{EP_j,k}$ to Ω_i solves

$$\int_{\Omega_i} \nabla\Psi_{EP_j,k} \cdot \nabla v \, \mathrm{d}\boldsymbol{x} = 0, \quad \forall v \in H_0^1(\Omega_i)$$

$$\Psi_{EP_j,k} = \mathcal{I}_j(\chi_k), \quad \text{on } EP_j$$

$$\Psi_{EP_j,k} = 0, \quad \text{on } \partial\Omega_i \setminus EP_j.$$

Note that when EP_j is adjacent to two components, though the two parts of $\Psi_{EP_j,k}$ are defined separately, $\Psi_{EP_j,k}$ retains continuity due to the Dirichlet boundary condition along EP_j.

We define the harmonic extensions for corner port modes similarly. We may define the

extensions for the corner ports directly as the usual finite element piecewise linear "hat" functions, with each component being treated as an element. Thus $\Psi_{CP_j,1}$ is 1 at CP_j and decreases linearly along each adjacent edge port before reaching the value of 0 at the adjacent corner ports. Indeed, the restriction of $\Psi_{CP_j,1}$ to a single adjacent component is the harmonic extension of a simple linear function onto the component. Because of the way it is defined, $\Psi_{CP_j,1}$ will be supported only on the components adjacent to CP_j. We include the subscript "1" in the interest of consistency with the notation used for the extensions of edge ports. The inspiration for the corner ports and their extensions comes from [46].

Our approximation subspace, $V_R^c \subset V^c$, is defined as

$$V_R^c = \left(\bigoplus_{j=1}^{16} \operatorname{span} \left\{ \Psi_{CP_j,1} \right\} \right) \oplus \left(\bigoplus_{j=1}^{24} \operatorname{span} \left\{ \Psi_{EP_j,1}, \ldots, \Psi_{EP_j,n_e} \right\} \right)$$

with n_e modes for each port.

Detailed training algorithm to generate $\{\chi_k\}_{k=1}^{n_e}$

This subsection provides additional details for generating the trace snapshots. On the first reading the reader may wish to skip this section or to skim Algorithm 2 and then move on to the next section.

Let $\ell(a, b)$ be a linear function which evaluates to a and b at the left and right endpoints, respectively, of the domain over which it is defined. Given a random variable, r, uniformly distributed on $[-1, 1]$, our training procedure is summarized in Algorithm 2.

A few remarks need to be made about our training procedure. First, $\tilde{C}_k^{(\alpha)}$ refers to the degree k Gegenbauer polynomial with parameter α, multiplied by $(1 - x^2)$ and then mapped to the interval corresponding to the appropriate port. The polynomials $\tilde{C}_k^{(\alpha)}$ were chosen because they evaluate to zero at $x = \pm 1$ and are orthogonal with respect to the L^2 inner product

$$(f, g) = \int_{-1}^{1} f(x) g(x) \, \mathrm{d}x.$$

Since they vanish at their intervals of definition we have the freedom to select the values u^h must take at the points where ports along the boundary meet. To this end we select

Algorithm 2: Four-component port mode training

$portSnapshots = \emptyset$

for $n = 1, 2, \ldots, N_{samples}$ **do**

  Generate random parameters $\boldsymbol{\mu} \in \mathcal{D}$

  for $i = 1, 2, \ldots, 8$ **do**

    Assign random boundary conditions along γ_i:

$$z^h\big|_{\gamma_i} = \ell(a_i, b_i) + \sum_{k=0}^{N_{deg}} r \frac{1}{(k+1)^2} \tilde{C}_k^{(2.5)}$$

  end

  Solve (5.2) on T

  for $j = 1, 2, 3, 4$ **do**

    $portSnapshots \leftarrow portSnapshots \cup z^h\big|_{E_j}$

  end

end

some of these values, a_i and b_i, randomly, and choose the others to ensure that the boundary conditions are continuous. Our training procedure departs from the one of Eftang and Patera in that we use a four-component domain for training rather than one consisting of two-components.

Because we allow up to four components to meet at a single point and because of the way we will define $p(\boldsymbol{\mu})$ in our numerical experiments, it is possible for singularities to arise that are avoided when at most two components meet at a port. In order to induce such singularities and capture their behavior in our snapshots we require a four-component training domain. Depending on the nature of the coefficient function, p, it may be prudent to separate the snapshots from ports E_1 and E_3 from those from E_2 and E_4. It may not even be possible to concatenate all the snapshots together if there are different numbers of degrees of freedom associated with vertical and horizontal ports (e.g. if the mesh spacing is different in the x and y directions).

Once the solution trace snapshots have been collected in Algorithm 2 we perform one extra preprocessing step: we remove the values of each snapshot at the two endpoints by subtracting a linear function. The result is a set of modified 1D snapshots which are zero at both endpoints. To these modified snapshots we apply the POD, which produces a set of N_e L^2-orthonormal functions which are zero at both endpoints and which form a basis for each edge port, EP_j. We refer to these functions as *port modes*. From this point forward when we shall assume that the basis defined on EP_j, $\{\mathcal{I}_j(\chi_k)\}_{k=1}^{N_e}$ consists of port modes obtained through this process. In some instances we will use other bases for EP_j than those obtained via the POD. In such cases we will refer to the POD modes as *empirical* port modes, as they are computed as the result of a training process.

Finally we truncate the edge port bases and retain n_e port modes, according to the following criterion on singular values

$$\frac{\sigma_k}{\sigma_1} \geq TOL,$$

where TOL is a small parameter. In the experiments of Section 5.4 we take $TOL = 3 \times 10^{-15}$.

5.3.2 Static condensation reduced basis element method

Here we build a subspace of $X^h \cap \tilde{H}_0^1(\Omega_i)$. We will refer to

$$X^h \cap \tilde{H}_0^1(\Omega_i) = \left\{ v \in X^h; v|_{\Omega \setminus \Omega_i} = 0 \right\}$$

as the *bubble* space associated with Ω_i. The bubble functions in these spaces will encapsulate the degrees of freedom in the interior of the components.

To simplify the upcoming discussion, we impose a global ordering on all the ports by concatenating them together. Let $P_1, P_2, \ldots, P_{40}$ be the set of all the ports of Ω ($P_i = CP_i$ for $i = 1, 2, \ldots, 16$ and $P_{i+16} = EP_i$ for $i = 1, 2, \ldots, 24$). Similarly to Eftang and Patera [37, p. 6], we define the energy-minimizing extension[3] $\Phi_{P_j,k}(\boldsymbol{\mu}) \in X^h$ as

$$\begin{aligned} a\left(\Phi_{P_j,k}(\boldsymbol{\mu}), v^h; \boldsymbol{\mu}\right) &= 0, \quad \forall \\ v^h \in \bigoplus_{i=1}^{9} X^h \cap \tilde{H}_0^1(\Omega_i) & \\ \Phi_{P_j,k}(\boldsymbol{\mu}) &= \mathcal{I}_j(\chi_k), \quad \text{on } P_j \\ \Phi_{P_j,k}(\boldsymbol{\mu}) &= 0, \quad \text{elsewhere on } \Sigma. \end{aligned} \tag{5.3}$$

On each component Ω_i, we define the bubble function, $b_{j,i,k}$, such that

$$b_{j,i,k}(\boldsymbol{\mu}) \in X^h \cap \tilde{H}_0^1(\Omega_i) \quad \text{and} \quad \Phi_{P_j,k}(\boldsymbol{\mu}) = b_{j,i,k}(\boldsymbol{\mu}) + \Psi_{P_j,k} \quad \text{in } \Omega_i \tag{5.4}$$

(namely, the bubble function $b_{j,i,k}$ is the difference between the energy-minimizing extension and the harmonic extension).

In order to find each bubble function, $b_{j,i,k}(\boldsymbol{\mu})$, we must solve a problem of size $\mathcal{O}(h^{-2})$ during the online phase, since $\boldsymbol{\mu}$ is not known in advance. To reduce this online cost, we replace the bubble functions with a POD-based approximation,

$$b_{j,i,k}(\boldsymbol{\mu}) \approx \tilde{b}_{j,i,k}(\boldsymbol{\mu}) \in V_{R,i}$$

[3] We caution the reader that a distinction is drawn between the harmonic extension, which is specific to the Laplace equation, and the energy-minimizing extension, which is specific to the bilinear form, a in (5.1).

where $V_{R,i}$ is a subspace of $X^h \cap \tilde{H}_0^1(\Omega_i)$ that does not depend on $\boldsymbol{\mu}$.

The resulting approximation subspace in X^h is

$$\left[\bigoplus_{i=1}^{9} V_{R,i}\right] \oplus V_R^c \tag{5.5}$$

Detailed training algorithm to generate $V_{R,i}$

This subsection describes in detail how a basis for $V_{R,i}$ is computed. To simplify the discussion, we introduce a set of local indices for the ports of Ω_i, see Figure 5.5.

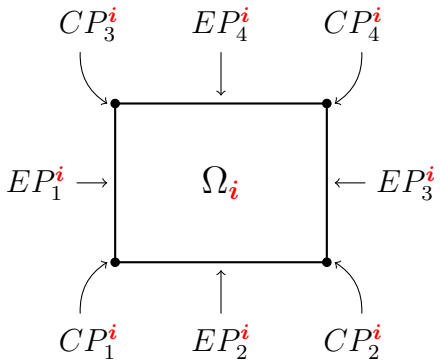


Figure 5.5: The local labeling of the ports for component Ω_i

The basis for $V_{R,i}$ is constructed from four PODs; one corresponding to each edge port. Observe that any interface function induced by an edge port adjacent to the component, $\Phi_{EP_j^i,k}$, $j = 1, 2, 3, 4$, has nonzero trace along exactly one port, namely EP_j^i. As a result, we expect that the excitation for the corresponding bubble function will be localized near this port. Therefore we expect a bubble function associated with port EP_j^i to be more economically represented by POD modes specific to port EP_j^i than more general POD modes associated with the entire component.

For now we group the bubble functions corresponding to each of the corner ports with one of the sets of edge port bubble functions. We suspect that treating the corner points separately may result in an improvement in performance, but have yet to implement such a

change.

To compute the necessary PODs we require snapshots of bubble functions. These snapshots are obtained with another training procedure using a single-component domain $B = (0,1) \times (0,1)$, with ports as in Figure 5.5. We denote the restriction of $\Psi_{P_j,k}$ to an adjacent component, Ω_i, as $\psi_{P_j,i,k}$. Given a set of parameter values, $\boldsymbol{\mu}$, and a harmonic extension $\psi_{P_j,1,k}$ of port mode $\mathcal{I}_{P_j}(\chi_k)$, we compute a snapshot, b, by solving the following problem: Find $b_{P_j,k}(\boldsymbol{\mu}) \in X^h \cap \tilde{H}_0^1(B)$ such that

$$\begin{aligned} a(b_{P_j,k}(\boldsymbol{\mu}), v; \boldsymbol{\mu}) &= -a(\psi_{P_j,1,k}, v; \boldsymbol{\mu}) \quad \forall v \in X^h \cap \tilde{H}_0^1(B) \\ b_{P_j,k}(\boldsymbol{\mu}) &= 0 \quad \text{on } \partial B \end{aligned} \tag{5.6}$$

Note that one of the indices of the local harmonic extension is fixed at 1. This is because B consists of only one component. We drop this index when describing the resulting bubble function. We provide an overview of our training process in Algorithm 3

Algorithm 3: One-component bubble function training

```
bubbleSnapshots = ∅
for n = 1, 2, ..., N_bubbleSamples do
    Generate a set of parameter values μ
    for j = 1, 2, 3, 4 do
        Solve (5.6) using ψ_{CP_j,1,1} to obtain b_{CP_j,1}
        bubbleSnapshots ← bubbleSnapshots ∪ b_{CP_j,1}
        for k = 1, 2, ..., n_e do
            Solve (5.6) using ψ_{EP_j,1,k} to obtain b_{EP_j,k}
            bubbleSnapshots ← bubbleSnapshots ∪ b_{EP_j,k}
        end
    end
end
```

We have a choice of how to generate parameter values for each iteration of the outermost

loop. The values could be drawn from some distribution on $\mathcal{D}$ or decided deterministically. We have found that, for the problems we consider in Section 5.4, the practice producing the best results is to take regularly spaced points in each dimension of $\mathcal{D}$.

Upon successful collection of the snapshots we perform four PODs and select a number of modes after which to truncate the POD bases using the criteria

$$\frac{\sigma_k}{\sigma_1} \geq TOL.$$

The union of these four POD bases forms the basis for $V_{R,i}$. Since each component may be identified with $(0,1) \times (0,1)$, we may re-use this basis for each $V_{R,i}$, with $i = 1, 2, \ldots, 9$.

5.3.3 Summary

Here we introduce the discrete system that is solved to compute our approximation and give a high-level overview of the steps taken in the offline and online phases of our method.

Let $\boldsymbol{K}(\boldsymbol{\mu}) \in \mathbb{R}^{\mathcal{N}\times\mathcal{N}}$ and $\boldsymbol{f}(\boldsymbol{\mu}) \in \mathbb{R}^{\mathcal{N}}$ be the finite element stiffness matrix and right-hand side vector associated with (5.2). It follows that $\boldsymbol{u}^h(\boldsymbol{\mu})$ is obtained as the solution to

$$\boldsymbol{K}(\boldsymbol{\mu})\boldsymbol{u}^h(\boldsymbol{\mu}) = \boldsymbol{f}(\boldsymbol{\mu}). \tag{5.7}$$

Recall that there are 16 corner ports and 24 edge ports. Since each corner port is associated with one energy-minimizing extension and each edge port has n_e corresponding energy-minimizing extensions, the total number of such extensions is $\mathcal{N}_R = 16 + 24n_e$. Presumably, $\mathcal{N}_R \ll \mathcal{N}$. Let $\boldsymbol{\Phi}(\boldsymbol{\mu}) \in \mathbb{R}^{\mathcal{N}\times\mathcal{N}_R}$ be the matrix obtained by expressing each energy-minimizing extension, $\Phi_{P_j,k}(\boldsymbol{\mu})$, as a (size $\mathcal{N}$) column vector of its pointwise values and concatenating them together. To compute the function in (5.5) approximating $u^h(\boldsymbol{\mu})$, we solve the following linear system

$$\boldsymbol{\Phi}(\boldsymbol{\mu})^\top \boldsymbol{K}(\boldsymbol{\mu})\boldsymbol{\Phi}(\boldsymbol{\mu})\boldsymbol{U}_R(\boldsymbol{\mu}) = \boldsymbol{\Phi}(\boldsymbol{\mu})^\top \boldsymbol{f}(\boldsymbol{\mu}). \tag{5.8}$$

The pointwise values of the approximation are then given by $\boldsymbol{u}_R^h(\boldsymbol{\mu}) = \boldsymbol{\Phi}(\boldsymbol{\mu})\boldsymbol{U}_R(\boldsymbol{\mu})$. $u_R^h(\boldsymbol{\mu})$ itself is a piecewise linear function. We refer to

$$\boldsymbol{K}_R(\boldsymbol{\mu}) = \boldsymbol{\Phi}(\boldsymbol{\mu})^\top \boldsymbol{K}(\boldsymbol{\mu})\boldsymbol{\Phi}(\boldsymbol{\mu})$$

as the *global projected stiffness matrix* and

$$\boldsymbol{f}_R(\boldsymbol{\mu}) = \boldsymbol{\Phi}(\boldsymbol{\mu})^\top \boldsymbol{f}(\boldsymbol{\mu})$$

as the *global projected right-hand side vector*. Note that the system (5.8) is of size $\mathcal{N}_R \times \mathcal{N}_R$, which has much smaller dimension than system (5.7).

$\boldsymbol{K}_R(\boldsymbol{\mu})$ can be efficiently assembled from one-component projected stiffness matrices, $\boldsymbol{K}_R^i(\boldsymbol{\mu})$,
$i = 1, 2, \ldots, 9$, corresponding to the local bilinear forms. These matrices may, in turn, be constructed as sums of what we call *local projected stiffness matrices*, which come from exploiting the bilinearity of the local forms, the representation of the energy minimizing extensions (5.4), and the affine parameter dependence of Assumption 1. We precompute the local projected stiffness matrices in the offline phase. We can perform similar steps to compute $\boldsymbol{f}_R(\boldsymbol{\mu})$ efficiently.

Given $\mathcal{D}$ and h, the **offline phase** is summarized in Algorithm 4. The **online phase**, outlined in Algorithm 5, is designed to run in a much smaller amount of time than the offline phase. It requires as input a set of parameters $\boldsymbol{\mu} \in \mathcal{D}$. Note that all but the last step of the online phase depend on $\mathcal{N}_R$ rather than $\mathcal{N}$.

Algorithm 4: Offline phase

1. Carry out the port reduction process of Section 5.3.1 to obtain a set of port mode
2. Lift the port modes to adjacent components via (5.3)
3. Execute the training procedure of Algorithm 3 to construct POD modes for the bubble functions
4. Compute the local projected stiffness matrices and local projected right-hand side vectors

Algorithm 5: Online phase

for $i = 1, \ldots, 9$ **do**
  Compute the approximations for all the bubble functions in Ω_i
  Form $\boldsymbol{K}_R^i(\boldsymbol{\mu})$ and $\boldsymbol{f}_R^i(\boldsymbol{\mu})$
end
Assemble $\boldsymbol{K}_R(\boldsymbol{\mu})$ and $\boldsymbol{f}_R$
Solve the global projected system $\boldsymbol{K}_R(\boldsymbol{\mu})\boldsymbol{U}_R(\boldsymbol{\mu}) = \boldsymbol{f}_R(\boldsymbol{\mu})$
If the pointwise values of $u_R^h(\boldsymbol{\mu})$ are needed, compute them using $\boldsymbol{U}_R(\boldsymbol{\mu})$

5.4 Numerical experiments

In this section we present the outcomes of a number of numerical experiments. To simplify the implementation of this method we work with square components (the components may be identified with $(0,1) \times (0,1)$) and with square linear elements (Q1 elements). The finite element parameter, h, is the width of a single element. For each experiment we set $N_{samples} = 200$ in Algorithm 2 and, unless otherwise noted, $h = \frac{1}{72}$.

5.4.1 Piecewise constant coefficient function

Our first examples are similar to Examples 1 and 2 of [37]. We take the coefficient function, p, to be constant on each component

$$p(\boldsymbol{x}; \boldsymbol{\mu}) = \mu_i \quad \boldsymbol{x} \in \Omega_i.$$

In terms of Assumption 2, p_0 is a constant function, μ_i. Note that when p is constant in a component, the energy-minimizing extension is identical to the harmonic extension (i.e. for the Laplace equation). So, for this case, our method does not require any bubble functions.

Example 5.4.1 (Square domain)**.** For this experiment we let $\Omega = (0,3) \times (0,3)$, see Figure 5.6. Each component, Ω_i, is easily identified with $(0,1) \times (0,1)$. Nine parameters, μ_i, are used

and are chosen from the set $\mathcal{D} = [0.1, 10]^9$. Furthermore, this coefficient function permits us to decompose (5.2) in a simple manner: for any $w, v \in X^h$

$$a(w, v; \boldsymbol{\mu}) = \sum_{i=1}^{9} a_i(w|_{\Omega_i}, v|_{\Omega_i}; \boldsymbol{\mu}) = \sum_{i=1}^{9} \mu_i \int_{\Omega_i} \nabla w \cdot \nabla v \, \mathrm{d}\boldsymbol{x}.$$

This decomposition is exploited to speed up the construction of the projected stiffness matrix.

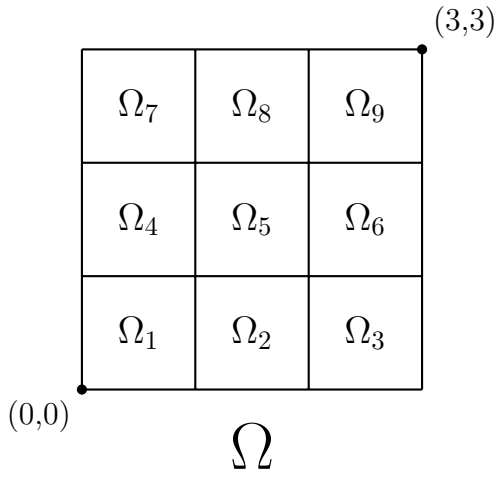


Figure 5.6: The domain, Ω, for Experiments 5.4.1, 5.4.3, and 5.4.4.

We choose $\Gamma_D = [0, 3] \times \{0\}$ and $\Gamma_N = \partial\Omega \setminus \Gamma_D$ and set

$$g(x, y; \boldsymbol{\mu}) = \begin{cases} \mu_7 & (x, y) \in [0, 1] \times \{3\} \\ 0 & \text{otherwise} \end{cases},$$

which corresponds to homogeneous Dirichlet boundary conditions along Γ_D, and Neumann boundary conditions along Γ_N.

Before we discuss the results of our numerical experiments, some remarks are in order. First, our choice of coefficient function departs from the one considered in [37]. Eftang and Patera worked with components which were cross-shaped (see Figure 5.7) to prevent more than two components from sharing a port. They used a coefficient function which was equal to one near the ports and μ_i in the interior of component Ω_i. This choice moves the

discontinuities of p away from the ports and into the components. Our function is defined in such a way that its discontinuities occur precisely along the ports.

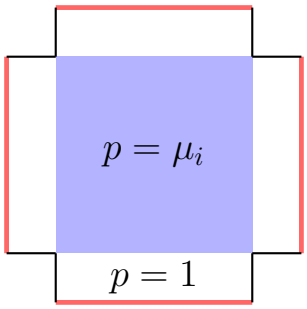


Figure 5.7: A cross-shaped component used by Eftang and Patera in [37]. In the blue region the coefficient function is identically μ_i and in the white region it is unity. The four ports are shown in red.

First we solve (5.2) with the particular parameter values used in Example 1 of [37], $\boldsymbol{\mu} = (0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4, 0.1, 0.2)$. The solution is visualized in Figure 5.8. Note that singularities may occur at the corner ports of four adjacent components (see, for instance, the corner $CP_{10} = (1, 2)$). These singularities are more pronounced when large and small parameter values are arranged in a checkerboard pattern.

We apply our approximation method to this problem and compute the relative error in the infinity norm

$$\frac{\left\|u^h(\boldsymbol{\mu}) - u_R^h(\boldsymbol{\mu})\right\|_\infty}{\left\|u^h(\boldsymbol{\mu})\right\|_\infty} \tag{5.9}$$

as we vary the number of port modes used. We also test the relative error (5.9) for two other possible choices of port modes: sinusoidal functions $\{\sin(k\pi x)\}_{k=1}^{N_e}$ and polynomials $\left\{\tilde{C}_k^{(2.5)}\right\}_{k=1}^{N_e}$. Figure 5.9 summarizes our results.

The empirical modes (circles) are much more economical for approximating the behavior of the solution along the edge ports than the sinusoidal or polynomial modes (squares and crosses, respectively). Roughly speaking, the relative error in our approximation decays at the same rate as the singular values from the POD. Data points with 0 port modes used refer to the relative error when only the corner port functions are employed.

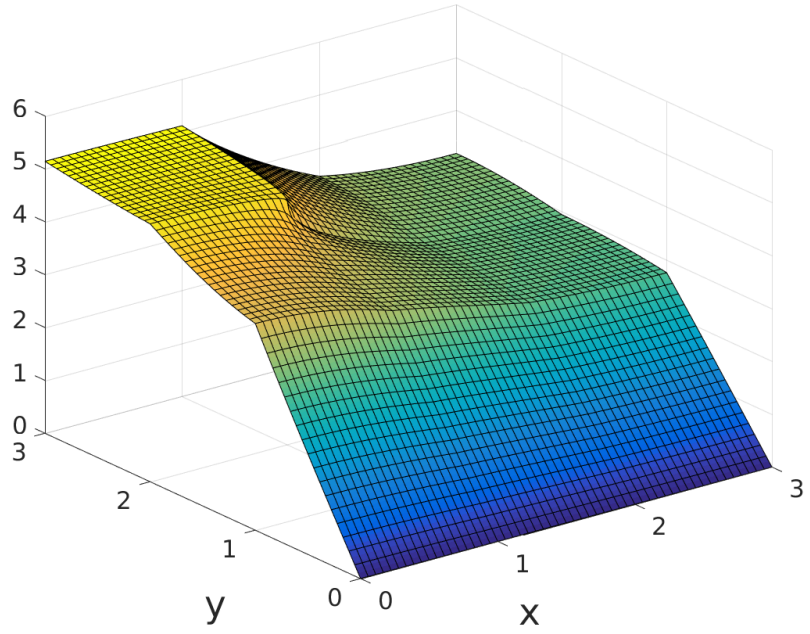


Figure 5.8: Example 5.4.1. The finite element solution for parameters $\boldsymbol{\mu} = (0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4, 0.1, 0.2)$.

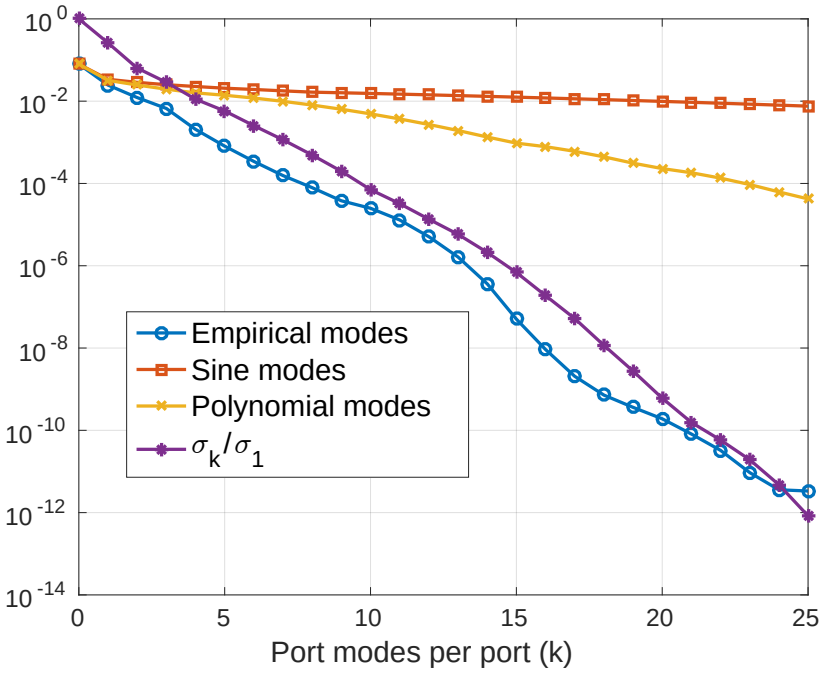


Figure 5.9: Example 5.4.1. A comparison of the relative error in the infinity norm for different choices of port modes (with $\boldsymbol{\mu} = (0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4, 0.1, 0.2)$) as the number of port modes, k, is varied. Rescaled singular values for the empirical port modes are also plotted.

This experiment only demonstrates the efficacy of our method for one particular set of parameters. We would also like to measure the robustness of our training procedure and port modes by computing the error for a variety of parameter values (in $\mathcal{D}$). To this end we generate 200 sets of random parameters and find the relative error in our approximation. The results of this test are plotted in Figure 5.10. The figure shows that for a fixed number of port modes (25) the performance of the method is fairly consistent across a range of parameter values. The median error was about 1.1×10^{-12}. Many of the parameter combinations generated during this test bring about singularities of the type pictured in Figure 5.8. Figure 5.10 suggests that the empirical port modes suggested in this work are well-equipped to deal with such singularities.

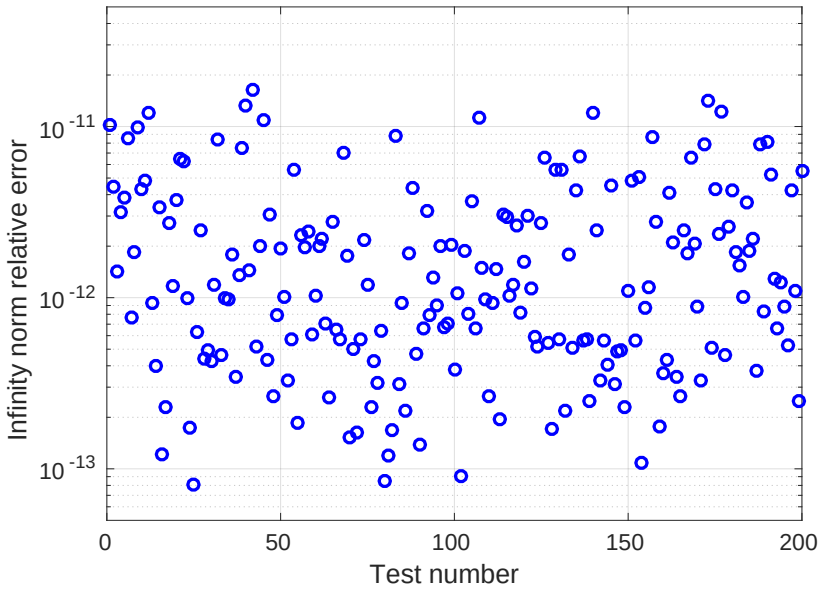


Figure 5.10: Example 5.4.1. The relative error in the infinity norm for 200 sets of random parameter samples using 25 port modes.

Example 5.4.2 (Horseshoe domain)**.** Next we consider a seven-component horseshoe-shaped domain, $\Omega \subset (0,3) \times (0,3)$, shown in Figure 5.11. Though the domain has changed, the individual components may still be identified with $(0,1) \times (0,1)$. The parameter set is now $\mathcal{D} = [0.1, 10]^7$. As in the previous example we may take advantage of the component-based

structure of Ω and p to break up the variational problem and reduce the complexity of forming the projected stiffness matrix.

We again take $\Gamma_D = [0,3] \times \{0\}$ and $\Gamma_N = \partial\Omega \setminus \Gamma_D$. For this experiment we take

$$g(x,y;\boldsymbol{\mu}) = \begin{cases} \mu_5 & (x,y) \in [0,1] \times \{3\} \\ 0 & \text{otherwise} \end{cases},$$

corresponding to homogeneous Dirichlet boundary conditions along Γ_D, $\frac{\partial u}{\partial n} = 1$ along $[0,1] \times \{3\} \subset \Gamma_N$, and homogeneous Neumann boundary conditions along the remainder of Γ_N.

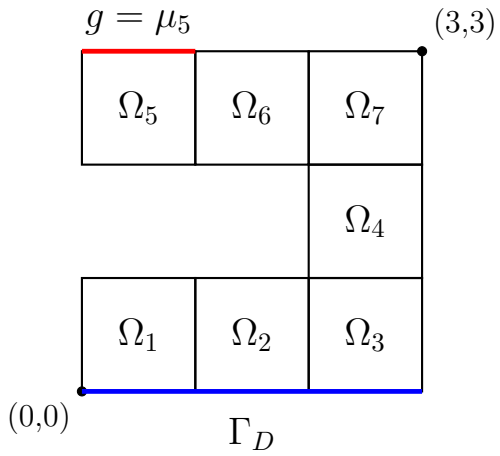


Figure 5.11: The horseshoe domain of Example 5.4.2

Letting $\boldsymbol{\mu} = (0.1, 0.2, 0.4, 3.2, 6.4, 0.1, 0.2)$ as in [37, Example 2] we compute the finite element solution, shown in Figure 5.12.

In Figure 5.13 we show the decay of the relative error as the number of port modes used in our method is allowed to grow. This curve can be compared to the rescaled singular values. As in the previous example we are able to obtain a reasonable accuracy using 25 port modes. Figure 5.14 plots the relative error in our method for 200 sets of randomly sampled parameters. Overall the performance of our method for this problem is similar to that seen in Example 5.4.1, albeit marginally worse. The median relative error across the random parameter test was about 6.2×10^{-12}.

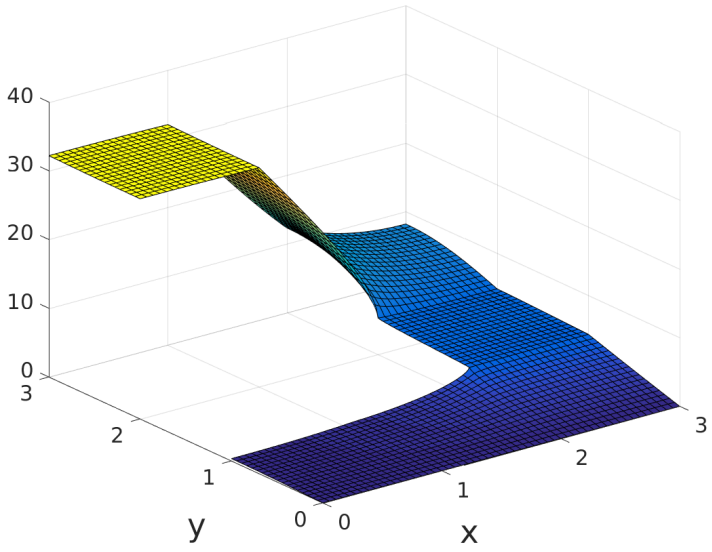


Figure 5.12: Example 5.4.2. The finite element solution for parameters $\boldsymbol{\mu} = (0.1, 0.2, 0.4, 3.2, 6.4, 0.1, 0.2)$.

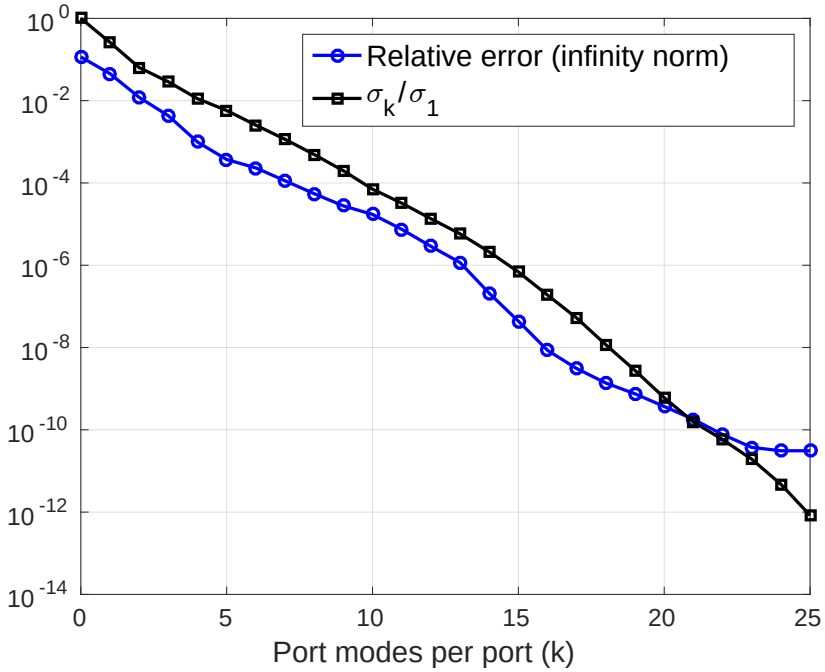


Figure 5.13: Example 5.4.2. The relative error in the infinity norm as the number of port modes, k, is varied for parameters $\boldsymbol{\mu} = (0.1, 0.2, 0.4, 3.2, 6.4, 0.1, 0.2)$.

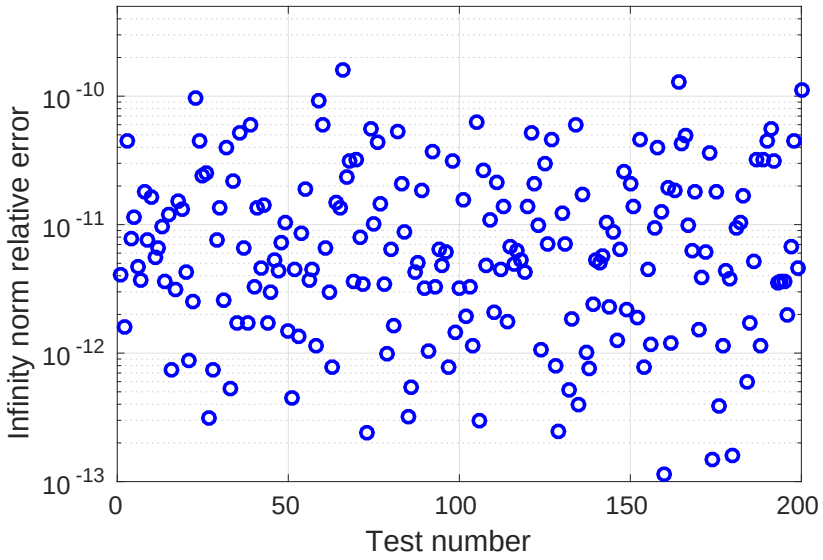


Figure 5.14: Example 5.4.2. The relative error in the infinity norm for 200 sets of random parameter samples using 25 port modes.

5.4.2 Cubic coefficient function

Next we consider two examples inspired by the problem studied in [7]. The domain will again be $\Omega = (0,3) \times (0,3)$. The coefficient function is chosen to be piecewise cubic (in x):

$$p_0^i(\xi, \eta; \tilde{\boldsymbol{\mu}}) = (\mu_{2i-1} + \xi\mu_{2i})^3.$$

We also enforce that p be continuous due to its interpretation in [7] as being an interpolant of a function that is continuous along the interface, Σ. Since each component has two parameters, it follows that $\boldsymbol{\mu} \in \mathbb{R}^{18}$, though we could reduce the dimension of $\boldsymbol{\mu}$ by noting that components with the same left and right endpoints share the same parameters and that the continuity condition on p imposes extra constraints. If we wished to be parsimonious we could take $\boldsymbol{\mu} \in \mathbb{R}^4$. We define $\mathcal{D}$ so that $\forall i$,

$$1 \times 10^{-5} \leq \mu_{2i-1} \leq 5 \times 10^{-5} \quad \text{and} \quad -4 \times 10^{-5} \leq \mu_{2i} \leq 4 \times 10^{-5}.$$

This choice of p satisfies Assumption 1 which greatly reduces the cost of assembling the linear system.

In the absence of further specification, the reader may assume that we set

$$p(x,y) = \left(5 - \frac{4}{3}x\right)^3 \times 10^{-5}$$

in the following experiments and that the POD approximation for the bubble functions is robust and accurate enough so that the error it introduces may be safely ignored. The examples differ from one another only in the boundary conditions that are imposed.

Example 5.4.3 (Singularity induced by boundary conditions)**.** First we let $\Gamma_N = \{3\} \times [0,1]$ and $\Gamma_D = \partial\Omega \setminus \Gamma_N$. To obtain unit Neumann boundary conditions along Γ_N, we set

$$g = (\mu_5 + 3\mu_6)^3 \qquad (\text{i.e.} g = p(3,y)).$$

We plot the finite element solution to this problem in Figure 5.15 (note the rotated perspective). Here a singularity is introduced due to the collision of homogeneous Dirichlet and unit Neumann boundary conditions at $(x,y) = (3,1)$. We shall see that this anomaly is not so easily handled by our method.

In Figure 5.16 we have the relative error as a function of the number of port modes used along with the singular values. The decay of the relative error does not match that of the singular values for this example. Rather, it is slower. This is due to the port modes facing difficulties resolving the solution along the ports adjacent to the singularity. Figure 5.17 visualizes the error in component three when 34 port modes are used on each edge port. Increasing the number of port modes, n_e, even further fails to improve the accuracy by a meaningful amount. We will need to make modifications to our method in order to improve its robustness in the presence of such difficulties.

As before, we benchmark our procedure by testing its accuracy for a large set of random parameters. Figure 5.18 plots the results of this test. Observe that in most cases the relative error is very close to the median, which is about 5.5×10^{-13}. However, there are some instances where the error is an order of magnitude or two worse. The unusually high inaccuracy in these cases can be attributed to deficiencies in the POD for the bubble functions. We were able to reduce the variance in relative error between tests by increasing the number of

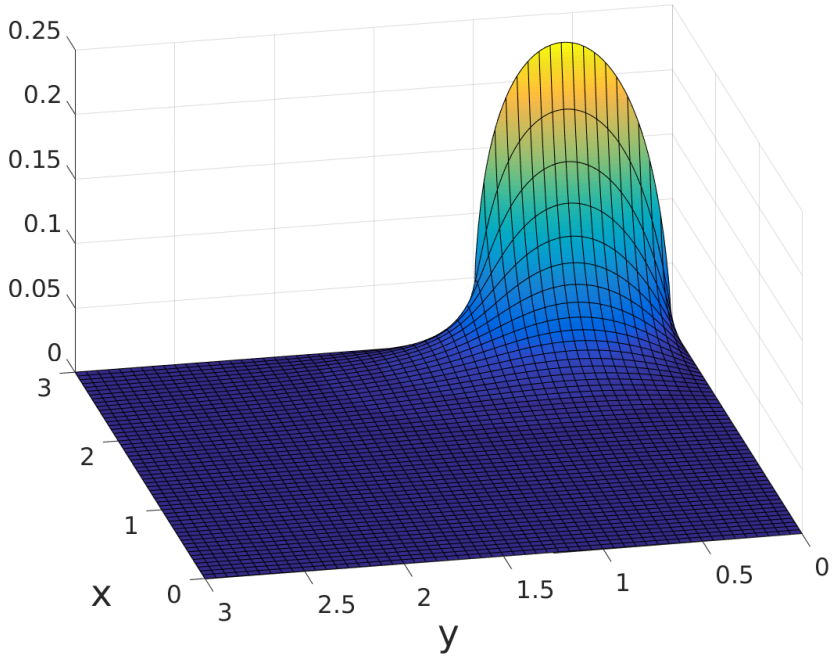


Figure 5.15: Example 5.4.3. The finite element solution. Note that the plot has been rotated so that the singularity at $(x, y) = (3, 1)$ is more visible.

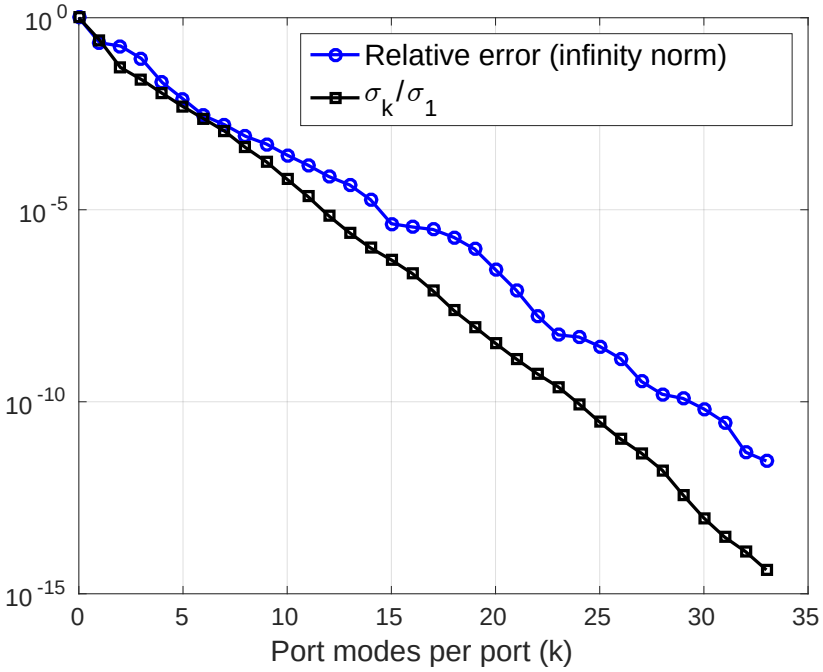


Figure 5.16: Example 5.4.3. The relative error in the infinity norm as the number of port modes, k, is varied and the scaled singular values associated with the port modes. The sequence $\left(\frac{\sigma_k}{\sigma_1}\right)^q$ matches the relative error for $q \approx 0.816$

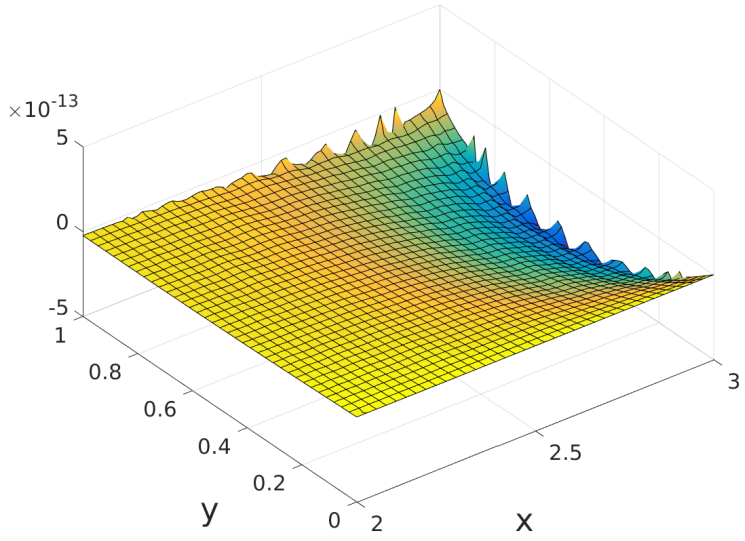


Figure 5.17: Example 5.4.3. A spatial plot of the difference between the finite element solution and our approximation (using 34 port modes) over Ω_3. Recall that Ω_3 is the component in the bottom-left of the square domain pictured in Figure 5.6, i.e. $\Omega_3 = (2,3) \times (0,1)$.

POD modes used in the bubble function approximation at the cost of a slower online phase. However, we are reasonably confident that we have a way to solve this problem by forming separate PODs for the bubble functions corresponding to interface functions for corner ports. Whereas we had previously included snapshots from the corner port interface functions in the bubble function POD, we would instead perform separate PODs on the edge and corner interface function snapshots. Assuming that we use one of these two strategies to improve the bubble function approximation, the limiting factor in terms of accuracy becomes the selection of an appropriate approximate basis for the edge ports. This problem is especially challenging when singularities in the solution appear due to boundary conditions, as this example illustrates.

Example 5.4.4 (Inhomogeneous Dirichlet boundary conditions)**.** For our last experiment we set unit Neumann boundary conditions along the entire right side of Ω by letting $\Gamma_N = \{3\} \times [0,3]$ and taking $g = (\mu_5 + 3\mu_6)^3$ (recall that components 3, 6, and 9 all share the

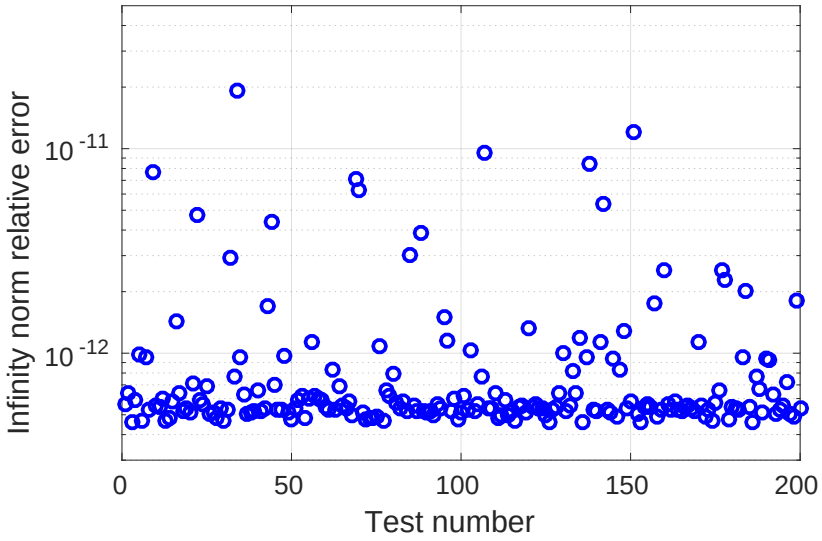


Figure 5.18: Example 5.4.3. The relative error in the infinity norm for 200 sets of random parameter samples using 34 port modes.

same parameter values). We introduce inhomogeneous Dirichlet boundary conditions along the top of Ω, $[0,3] \times \{3\}$, and retain homogeneous Dirichlet boundary conditions along the rest of $\partial\Omega$, $(\{0\} \times [0,3]) \cup ([0,3] \times \{0\})$. Along $[0,3] \times \{3\}$ we set

$$u(x,y) = \left(3x + \sin\left(\pi^2 x\right)\right)(3-x)/3. \tag{5.10}$$

A plot of the finite element solution is shown in Figure 5.19 (left).

Figure 5.20 plots the infinity norm relative error in our approximation as a function of the number of port modes along with the (scaled) singular values. Though the first few (low frequency) modes are unable to capture the oscillatory behavior of the sine function, eventually the error begins to decay at a similar rate to the singular values. However, if we were to continue the plot beyond 34 port modes we would witness a sudden plateau in both the relative error and the singular values. This plateau is due to the inability of the empirical port modes to mimic the trace of the solution along Γ_D, as can be seen in Figure 5.19 (right), which shows the difference between the finite element solution and our approximation. The error is completely localized along Γ_D. Indeed, if we ignore the entire

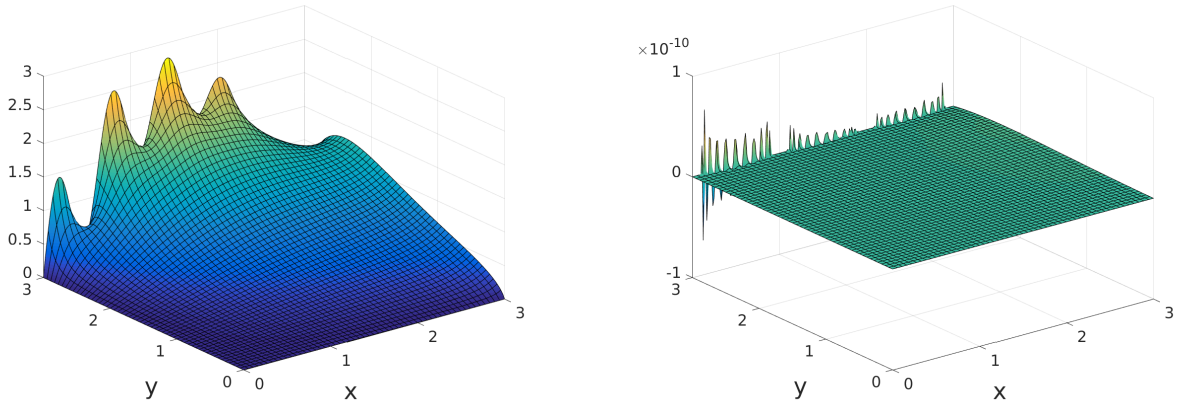


Figure 5.19: Example 5.4.4. Left: The finite element solution. Right: A spatial plot of the difference between the finite element solution and our approximation (using 34 port modes).

variational problem and simply compute the (infinity norm) relative error in the projection of the Dirichlet boundary condition, (5.10), onto the port modes, we obtain the piecewise linear curve in Figure 5.20. The curve matches up almost exactly with the relative error for the approximate solution.

A natural idea would be to use a different approximate basis for the edge ports which is better able to resolve the Dirichlet boundary condition. However, there is a tradeoff to be taken into consideration. The better a set of modes is at representing a general class of functions, the worse it may become at mimicking the specific behavior of the solution along the edge ports that are not a part of Γ_D. The empirical port modes are all taken directly from the solution manifold of the trace of the solution along an edge port, while functions chosen *a priori* are not likely to be members of this set. Therefore, upon replacing the empirical port modes with functions well-suited to approximate the Dirichlet boundary conditions, we would expect a degradation in the quality of the final approximation in the interior of Ω. Figure 5.21 provides some evidence in favor of our suspicion. For three sets of port modes (empirical, polynomial, and sinusoidal) it shows both the relative error in the infinity norm of the overall approximation and the relative error in the projection of (5.10)

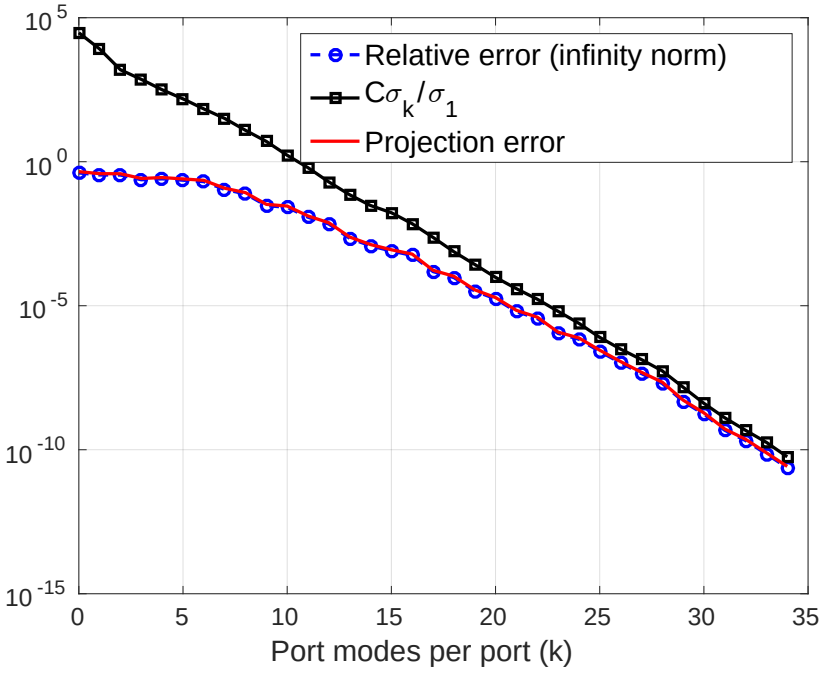


Figure 5.20: Example 5.4.4. The relative error in the infinity norm (circles), the scaled singular values associated with the port modes (squares), and the relative error in the projection of the Dirichlet boundary condition onto the port modes (line) as the number of port modes is varied.

onto the modes themselves. The projection error for the polynomial basis (downward-facing triangles) decays rapidly to machine precision, but its overall relative error (upward-facing triangles) decreases slowly. This shows that while this basis is adroit at replicating the Dirichlet boundary condition, it does a poor job of mirroring the solution along edge ports in the interior of Ω. The sinusoidal basis exhibits underwhelming performance in both cases.

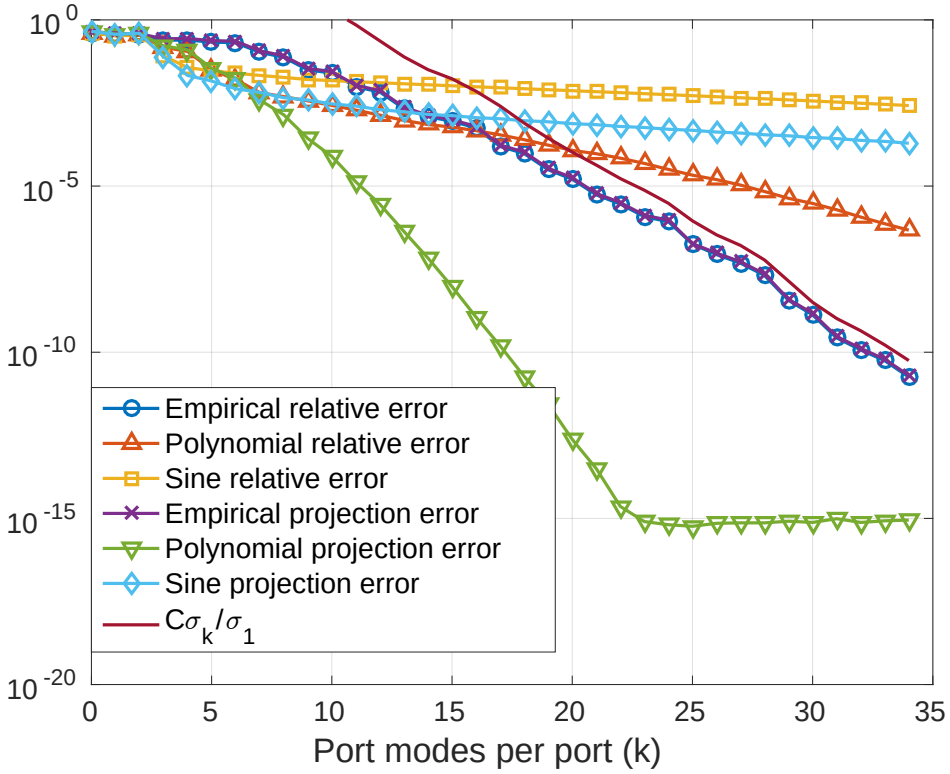


Figure 5.21: Example 5.4.4. A comparison of the relative error and relative error in the projection of the Dirichlet boundary condition onto the port modes for different choices of port modes as the number of port modes is varied. The scaled singular values for the empirical modes are also plotted.

The last two examples show that while the empirical port modes are adept at capturing the solution behavior along ports in the interior of Ω, more work is needed to enable them to accurately approximate the solution along the boundaries when challenging boundary conditions are imposed.

BIBLIOGRAPHY

[1] Elmar Achenbach. Experiments on the flow past spheres at very high reynolds numbers. *Journal of Fluid Mechanics*, 54(3):565–575, 1972.

[2] Elmar Achenbach. Vortex shedding from spheres. *Journal of Fluid Mechanics*, 62(2):209–221, 1974.

[3] Carl G Adler and Byron L Coulter. Galileo and the tower of pisa experiment. *American Journal of Physics*, 46(3):199–201, 1978.

[4] David J Allerton and Huamin Jia. A review of multisensor fusion methodologies for aircraft navigation systems. *The Journal of Navigation*, 58(3):405–417, 2005.

[5] Aleksandr Aravkin, James V Burke, and Gianluigi Pillonetto. Robust and trend-following kalman smoothers using student's t. *IFAC Proceedings Volumes*, 45(16):1215–1220, 2012.

[6] Aleksandr Y Aravkin, James V Burke, and Gianluigi Pillonetto. Optimization viewpoint on kalman smoothing with applications to robust and sparse estimation. In *Compressed sensing & sparse filtering*, pages 237–280. Springer, 2014.

[7] Eduard Bader, Martin A Grepl, and Siegfried Müller. A static condensation reduced basis element approach for the Reynolds lubrication equation. *Communications in Computational Physics*, 21(1):126–148, 2017.

[8] DF Bartlett, PE Goldhagen, and EA Phillips. Experimental test of coulomb's law. *Physical Review D*, 2(3):483, 1970.

[9] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510, 2016.

[10] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[11] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[12] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

[13] Josh Bongard and Hod Lipson. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 104(24):9943–9948, 2007.

[14] Lorenzo Boninsegna, Feliks Nüske, and Cecilia Clementi. Sparse learning of stochastic dynamical equations. *The Journal of chemical physics*, 148(24):241723, 2018.

[15] Frédéric Bourquin. Component mode synthesis and eigenvalues of second order operators: discretization and algorithm. *ESAIM: Mathematical Modelling and Numerical Analysis*, 26(3):385–423, 1992.

[16] Leo Breiman et al. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231, 2001.

[17] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. Classification and regression trees. wadsworth int. *Group*, 37(15):237–251, 1984.

[18] Susanne Brenner and Ridgway Scott. *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media, 2007.

[19] Will Bridewell, Pat Langley, Ljupčo Todorovski, and Sašo Džeroski. Inductive process modeling. *Machine learning*, 71(1):1–32, 2008.

[20] Phillip P Brown and Desmond F Lawler. Sphere drag and settling velocity revisited. *Journal of environmental engineering*, 129(3):222–231, 2003.

[21] S. L. Brunton, J. L. Proctor, and J. N. Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[22] Steven L. Brunton, Bingni W. Brunton, Joshua L. Proctor, Eurika Kaiser, and J. Nathan Kutz. Chaos as an intermittently forced linear system. *Nature Communications*, 8(1), December 2017.

[23] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, page 201517384, 2016.

[24] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Sparse identification of nonlinear dynamics with control (sindyc)**slb acknowledges support from the u.s. air force center of excellence on nature inspired flight technologies and ideas (fa9550-14-1-0398). jlp thanks bill and melinda gates for their active support of the institute of disease modeling and their sponsorship through the global good fund. jnk acknowledges support from the u.s. air force office of scientific research (fa9550-09-0174). *IFAC-PapersOnLine*, 49(18):710 – 715, 2016. 10th IFAC Symposium on Nonlinear Control Systems NOLCOS 2016.

[25] Andreas Buhr and Kathrin Smetana. Randomized local model order reduction. *arXiv preprint arXiv:1706.09179*, 2017.

[26] JR Calvert. Some experiments on the flow past a sphere. *The Aeronautical Journal*, 76(736):248–250, 1972.

[27] K. Champion, S. L. Brunton, and J. N. Kutz. Discovery of nonlinear multiscale systems: Sampling strategies and embeddings. 05 2018.

[28] Kathleen Champion, Peng Zheng, Aleksandr Y Aravkin, Steven L Brunton, and J Nathan Kutz. A unified sparse optimization framework to learn parsimonious physics-informed models from data. *arXiv preprint arXiv:1906.10612*, 2019.

[29] Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

[30] Rick Chartrand. Numerical differentiation of noisy, nonsmooth data. *ISRN Applied Mathematics*, 2011, 2011.

[31] Rasmus S Christensen, Ricky Teiwes, Steffen V Petersen, Ulrik I Uggerhøj, and Bo Jacoby. Laboratory test of the galilean universality of the free fall experiment. *Physics Education*, 49(2):201, 2014.

[32] Lane Cooper. Aristotle, galileo, and the tower of pisa. 1936.

[33] Rod Cross and Crawford Lindsey. Measuring the drag force on a falling ball. *The Physics Teacher*, 52(3):169–170, 2014.

[34] Magnus Dam, Morten Brøns, Jens Juul Rasmussen, Volker Naulin, and Jan S Hesthaven. Sparse identification of a predator-prey system from simulation data of a convection model. *Physics of Plasmas*, 24(2):022310, 2017.

[35] Brian de Silva, David M Higdon, Steven L Brunton, and J Nathan Kutz. Discovery of physics from data: Universal laws and discrepancy models. *arXiv preprint arXiv:1906.07906*, 2019.

[36] Pedro Domingos. The role of occam's razor in knowledge discovery. *Data mining and knowledge discovery*, 3(4):409–425, 1999.

[37] Jens L Eftang and Anthony T Patera. Port reduction in parametrized component static condensation: approximation and a posteriori error estimation. *International Journal for Numerical Methods in Engineering*, 96(5):269–302, 2013.

[38] L. Van Eykeren and Q. P. Chu. Sensor fault detection and isolation for aircraft control systems by kinematic relations. *Control Engineering Practice*, 31:200–210, 2014.

[39] Isobel Falconer. No actual measurement... was required: Maxwell and cavendish's null method for the inverse square law of electrostatics. *Studies in History and Philosophy of Science Part A*, 65:74–86, 2017.

[40] Patrick Gelß, Stefan Klus, Jens Eisert, and Christof Schütte. Multidimensional approximation of nonlinear dynamical systems. *Journal of Computational and Nonlinear Dynamics*, 14(6), 2019.

[41] Diego Del Gobbo, Marcello Napolitano, Parviz Famouri, and Mario Innocenti. Experimental application of extended Kalman filtering for sensor validation. *IEEE Transactions on Control Systems Technology*, 9(2), 2001.

[42] John Eric Goff. A review of recent research into aerodynamics of sport projectiles. *Sports engineering*, 16(3):137–154, 2013.

[43] M. Goman and A. Khrabrov. State-space representation of aerodynamic characteristics of an aircraft at high angles of attack. *Journal of Aircraft*, 31(5):1109–1115, 1994.

[44] P. Goupil and A. Marcos. Advanced diagnosis for sustainable flight guidance and control: the European ADDSAFE project. Technical paper 2011-01-2804, SAE, 2011.

[45] Chingiz Hajiyev. Testing the covariance matrix of the innovation sequence with sensor/actuator fault detection applications. *International Journal of Adaptive Control and Signal Processing*, 24:717–730, 2010.

[46] Ulrich L Hetmaniuk and Richard B Lehoucq. A special finite element method based on component mode synthesis. *ESAIM: Mathematical Modelling and Numerical Analysis*, 44(3):401–420, 2010.

[47] Moritz Hoffmann, Christoph Fröhner, and Frank Noé. Reactive SINDy: Discovering governing reactions from concentration data. *Journal of Chemical Physics*, 150(025101), 2019.

[48] Dinh Bao Phuong Huynh, David J Knezevic, and Anthony T Patera. A static condensation reduced basis element method: approximation and a posteriori error estimation. *ESAIM: Mathematical Modelling and Numerical Analysis*, 47(1):213–251, 2013.

[49] Mihailo R. Jovanović, Peter J. Schmid, and Joseph W. Nichols. Sparsity-promoting dynamic mode decomposition. *Physics of Fluids*, 26, 2014.

[50] J. N. Juang and R. S. Pappa. An eigensystem realization algorithm for modal parameter identification and model reduction. *J. of Guidance, Control, and Dynamics*, 8(5):620–627, 1985.

[51] J. N. Juang, M. Phan, L. G. Horta, and R. W. Longman. Identification of observer/Kalman filter Markov parameters: theory and experiments. Technical Memorandum 104069, NASA, 1991.

[52] Chakkrit Kaewsutthi and Pornrat Wattanakasiwich. Student learning experiences from drag experiments using high-speed video analysis. In *Proceedings of The Australian Conference on Science and Mathematics Education (formerly UniServe Science Conference)*, volume 17, 2011.

[53] Eurika Kaiser, J Nathan Kutz, and Steven L Brunton. Sparse identification of nonlinear dynamics for model predictive control in the low-data limit. *Proceedings of the Royal Society of London A*, 474(2219), 2018.

[54] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[55] Johann Kepler. Astronomia nova. *(Pragae) 1609*, 2015.

[56] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*, volume 149. SIAM, 2016.

[57] J Nathan Kutz, Joshua L Proctor, and Steven L Brunton. Koopman theory for partial differential equations. *arXiv preprint arXiv:1607.07076*, 2016.

[58] Zhilu Lai and Satish Nagarajaiah. Sparse structural system identification method for nonlinear dynamic systems with hysteresis/inelastic behavior. *Mechanical Systems and Signal Processing*, 117:813–842, 2019.

[59] J. G. Leishman. *Principles of Helicopter Aerodynamics*. Cambridge University Press, 2002.

[60] R. Li and J. H. Olson. Fault detection and diagnosis an a closed-loop nonlinear distillation process: Application of extended Kalman filters. *Industrial & Engineering Chemistry Research*, 30:898–908, 1991.

[61] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[62] J.-C. Loiseau and S. L. Brunton. Constrained sparse Galerkin regression. *Journal of Fluid Mechanics*, 838:42–67, 2018.

[63] J.-C. Loiseau, B. R. Noack, and S. L. Brunton. Sparse reduced-order modeling: sensor-based dynamics to full-state estimation. *Journal of Fluid Mechanics*, 844:459–490, 2018.

[64] Jean-Christophe Loiseau. Data-driven modeling of the chaotic thermal convection in an annular thermosyphon. *arXiv preprint arXiv:1911.07920*, 2019.

[65] Susan Lomax and Sunil Vadera. A survey of cost-sensitive decision tree induction algorithms. *ACM Computing Surveys (CSUR)*, 45(2):1–35, 2013.

[66] RH Magarvey and CS MacLatchy. Vortices in sphere wakes. *Canadian Journal of Physics*, 43(9):1649–1656, 1965.

[67] Niall M Mangan, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Inferring biological networks by sparse identification of nonlinear dynamics. *IEEE Transactions on Molecular, Biological, and Multi-Scale Communications*, 2(1):52–63, 2016.

[68] Niall M Mangan, J Nathan Kutz, Steven L Brunton, and Joshua L Proctor. Model selection for dynamical systems via sparse regression and information criteria. *Proceedings of the Royal Society A*, 473(2204):1–16, 2017.

[69] James Clerk Maxwell. *A treatise on electricity and magnetism*, volume 1. Oxford: Clarendon Press, 1873.

[70] R. K. Mehra and J. Peschon. An innovations approach to fault detection and diagnosis in dynamic systems. *Automatica*, 7(637-640), 1971.

[71] Rabindra D Mehta. Aerodynamics of sports balls. *Annual Review of Fluid Mechanics*, 17(1):151–189, 1985.

[72] Rabindra D Mehta. Sports ball aerodynamics. In *Sport Aerodynamics*, pages 229–331. Springer, 2008.

[73] MIL-HDBK-1797. *Flying qualities of piloted aircraft*. Dept. of Defense, 1990.

[74] W Moller. Experimentelle untersuchung zur hydromechanick der hugel, phys. *Z*, 35:57–80, 1938.

[75] Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li F Fei-Fei, Josh Tenenbaum, and Daniel L Yamins. Flexible neural representation for physics prediction. In *Advances in Neural Information Processing Systems*, pages 8799–8810, 2018.

[76] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[77] M. R. Napolitano. Online learning neural architectures and cross-correlation analysis for actuator failure detection and identification. *International Journal of Control*, 63(3):433–455, 1996.

[78] M. R. Napolitano, C. Chen, and S. Naylo. Aircraft failure detection and identification using neural networks. *Journal of Guidance, Control, and Dynamics*, 16(6):999–1009, 1993.

[79] John Newman. *Marine Hydrodynamics*. The MIT Press, Cambridge, Massachusetts, 1977.

[80] Isaac Newton. *The Principia: mathematical principles of natural philosophy*. Univ of California Press, 1999.

[81] Runhai Ouyang, Stefano Curtarolo, Emre Ahmetcik, Matthias Scheffler, and Luca M. Ghiringhelli. Sisso: A compressed-sensing method for identifying the best low-dimensional descriptor in an immensity of offered candidates. *Phys. Rev. Materials*, 2:083802, Aug 2018.

[82] Julia P Owen and William S Ryu. The effects of linear and quadratic drag on falling spheres: an undergraduate laboratory. *European Journal of Physics*, 26(6):1085, 2005.

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

[84] Christian Heinrich Friedrich Peters and Edward Ball Knobel. Ptolemy's catalogue of stars: A revision of the almagest. *Physics and Chemistry in Space*, 1915.

[85] Joshua L. Proctor, Steven L. Brunton, and J. Nathan Kutz. Dynamic mode decomposition with control. *SIAM Journal of Applied Dynamical Systems*, 15(1):142–161, 2016.

[86] Claudius Ptolemy. *The almagest: introduction to the mathematics of the heavens*. Green Lion Press, 2014.

[87] Markus Quade. sparsereg - collection of modern sparse regression algorithms, February 2018.

[88] Maziar Raissi. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *The Journal of Machine Learning Research*, 19(1):932–955, 2018.

[89] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using gaussian processes. *Journal of Computational Physics*, 348:683–693, 2017.

[90] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[91] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

[92] H. Raza, P. Ioannou, and H. M. Youssef. Surface failure detection for an F/A-18 aircraft using neural networks and fuzzy logic. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 5, pages 3363–3368, 1994.

[93] Patrick AK Reinbold, Daniel R Gurevich, and Roman O Grigoriev. Using noisy or incomplete data to discover models of spatiotemporal dynamics. *Physical Review E*, 101(1):010203, 2020.

[94] C. W. Rowley, I. Mezić, S. Bagheri, P. Schlatter, and D.S. Henningson. Spectral analysis of nonlinear flows. *J. Fluid Mech.*, 645:115–127, 2009.

[95] S. Rudy, A. Alla, S. L. Brunton, and J. N. Kutz. Data-driven identification of parametric partial differential equations. *SIAM Journal on Applied Dynamical Systems*, 18(2):643–660, 2019.

[96] S. H. Rudy, S. L. Brunton, J. L. Proctor, and J. N. Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(e1602614), 2017.

[97] Samuel Rudy, Alessandro Alla, Steven L Brunton, and J Nathan Kutz. Data-driven identification of parametric partial differential equations. *arXiv preprint arXiv:1806.00732*, 2018.

[98] Pratik Sachdeva, Jesse Livezey, Andrew Tritt, and Kristofer Bouchard. Pyuoi: The union of intersections framework in python. *Journal of Open Source Software*, 4(44):1799, 2019.

[99] Abraham. Savitzky and M. J. E. Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 36(8):1627–1639, 1964.

[100] Hayden Schaeffer. Learning partial differential equations via data discovery and sparse optimization. In *Proc. R. Soc. A*, volume 473, page 20160446. The Royal Society, 2017.

[101] Hayden Schaeffer and Scott G McCalla. Sparse model selection via integral terms. *Physical Review E*, 96(2):023302, 2017.

[102] Hayden Schaeffer, Giang Tran, and Rachel Ward. Extracting sparse high-dimensional dynamics from limited data. *SIAM Journal on Applied Mathematics*, 78(6):3279–3295, 2018.

[103] Peter J. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, 2010.

[104] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[105] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[106] Michael Segre. The role of experiment in galileo's physics. *Archive for History of Exact Sciences*, 23(3):227–252, 1980.

[107] Kathrin Smetana and Anthony T Patera. Optimal local approximation spaces for component-based static condensation procedures. *SIAM Journal on Scientific Computing*, 38(5):A3318–A3356, 2016.

[108] Alexander J Smits and Steven Ogg. Aerodynamics of the golf ball. In *Biomedical engineering principles in sports*, pages 3–27. Springer, 2004.

[109] Mariia Sorokina, Stylianos Sygletos, and Sergei Turitsyn. Sparse identification for nonlinear optical communication systems: SINO method. *Optics express*, 24(26):30433–30443, 2016.

[110] R. F. Stengel. *Flight Dynamics*. Princeton University Press, 2004.

[111] Weijie Su, Małgorzata Bogdan, Emmanuel Candes, et al. False discoveries occur early on the lasso path. *The Annals of statistics*, 45(5):2133–2150, 2017.

[112] A. Surana and A. Banaszuk. Linear observer synthesis for nonlinear systems using koopman operator framework. *IFAC-PapersOnLine*, 49(18):716–723, 2016.

[113] Amit Surana. Koopman operator framework for time series modeling and analysis. *Journal of Nonlinear Science*, 2018.

[114] Josué Sznitman, Howard A Stone, Alexander J Smits, and James B Grotberg. Teaching the falling ball problem with dimensional analysis. *European Journal of Physics Education*, 4(2):44–54, 2017.

[115] Floris Takens. Detecting strange attractors in turbulence. In *Dynamical systems and turbulence, Warwick 1980*, pages 366–381. Springer, 1981.

[116] Jovan Tanevski, Nikola Simidjievski, Ljupčo Todorovski, and Sašo Džeroski. Process-based modeling and design of dynamical systems. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 378–382. Springer, 2017.

[117] Jovan Tanevski, Ljupčo Todorovski, and Sašo Džeroski. Learning stochastic process-based models of dynamical systems from knowledge and data. *BMC systems biology*, 10(1):30, 2016.

[118] Stephan Thaler, Ludger Paehler, and Nikolaus A Adams. Sparse identification of truncation errors. *Journal of Computational Physics*, 397:108851, 2019.

[119] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[120] Giang Tran and Rachel Ward. Exact recovery of chaotic systems from highly corrupted data. *Multiscale Modeling & Simulation*, 15(3):1108–1129, 2017.

[121] J. H. Tu, C. W. Rowley, D. M. Luchtenburg, S. L. Brunton, and J. N. Kutz. On dynamic mode decomposition: theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

[122] Sylvain Vallaghé. The static condensation reduced basis element method for parabolic problems. In *M3AS: Math Models Methods Appl Sci*. 2013.

[123] Sylvain Vallaghé, Phuong Huynh, David J Knezevic, Loi Nguyen, and Anthony T Patera. Component-based reduced basis for parametrized symmetric eigenproblems. *Advanced Modeling and Simulation in Engineering Sciences*, 2(1):7, 2015.

[124] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri. A review of process fault detection and diagnosis: Part I: Quantitative model-based methods. *Computational Chemical Engineering*, 27(3):293–311, 2003.

[125] B. K. Walker and K. Huang. FDI by extended Kalman filter parameter estimation for the industrial actuator benchmark. In *IFAC Symposiym SAFE-PROCESS*, pages 481–487, 1994.

[126] S.-T. Wang and W. Frost. Atmospheric turbulence simulation techniques with application to flight analysis. Technical report, NASA, 1980.

[127] Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. Visual interaction networks: Learning a physics simulator from video. In *Advances in neural information processing systems*, pages 4539–4547, 2017.

[128] Frank M White and RY Chul. *Fluid Mechanics, 2011*. New-York, MacGraw-Hill, 2011.

[129] Matthew O. Williams, Ioannis G. Kevrekidis, and Clarence W. Rowley. A data–driven approximation of the Koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, December 2015.

[130] Matthew O Williams, Clarence W Rowley, and Ioannis G Kevrekidis. A kernel-based method for data-driven Koopman spectral analysis. *Journal of Computational Dynamics*, 2(2):247–265, 2015.

[131] A. S. Willsky. A survey of design methods for failure detection in dynamic systems. *Automatica*, 12(6):601–611, 1976.

[132] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[133] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[134] Peng Zheng, Travis Askham, Steven L Brunton, J Nathan Kutz, and Aleksandr Y Aravkin. A unified framework for sparse relaxed regularized regression: Sr3. *IEEE Access*, 7:1404–1423, 2018.