Czech Technical University in Prague

Faculty of Mechanical Engineering

Department of Instrumentation and Control Engineering

Master thesis

Model discovery from data using methods for sparse identification of nonlinear implicit dynamics

2020/2021

Kryštof Bystřický

Supervisor: Ing. Jaroslav Bušek, Ph.D.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bystřický** Jméno: **Kryštof** Osobní číslo: **465337**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Automatizační a přístrojová technika**

Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Identifikace modelů z dat pomocí metod řídké identifikace nelineární implicitní dynamiky

Název diplomové práce anglicky:

Model discovery from data using methods for sparse identification of nonlinear implicit dynamics

Pokyny pro vypracování:

1) Proveďte rešerši na téma metody řídké identifikace regrese pro identifikaci parametrů nelineárních implicitních modelů z dat a seznamte se s aplikačními aspekty této metody.
2) Popište metody předzpracování dat pro účely identifikace, algoritmy řídké regrese a metody validace a výběru modelů.
3) Zvolte vhodný nelineární model s racionálními funkcemi a aplikujte popisovanou metodu na identifikaci jeho parametrů. Proveďte validaci parametrů modelu.
4) Zhodnoťte dosažené výsledky.

Seznam doporučené literatury:

[1] K. Kaheman, J. N. Kutz, and S. L. Brunton, “SINDy-PI: A Robust Algorithm for Parallel Implicit Sparse Identification of Nonlinear Dynamics,” arXiv:2004.02322 [physics, stat], Sep. 2020, Accessed: Mar. 31, 2021. [Online]. Available: http://arxiv.org/abs/2004.02322.
[2] S. L. Brunton, J. L. Proctor, and J. N. Kutz, “Discovering governing equations from data by sparse identification of nonlinear dynamical systems,” PNAS, vol. 113, no. 15, pp. 3932–3937, Apr. 2016, doi: 10.1073/pnas.1517384113.
[3] S. L. Brunton and J. N. Kutz, Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control, 1st ed. Cambridge University Press, 2019.
[4] S. L. Brunton, J. L. Proctor, and J. N. Kutz, “Sparse Identification of Nonlinear Dynamics with Control (SINDYc),” arXiv:1605.06682 [math], May 2016, Accessed: Mar. 31, 2021. [Online]. Available: http://arxiv.org/abs/1605.06682.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jaroslav Bušek, Ph.D., U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **30.04.2021** Termín odevzdání diplomové práce: **24.08.2021**

Platnost zadání diplomové práce: ______________

Ing. Jaroslav Bušek, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Michael Valášek, DrSc. podpis děkana(ky)

Abstract

The problem of obtaining models describing the dynamics of a certain system is current across many industries. The traditional approach to modeling relies on specific knowledge about the physical principles of the system. This approach requires a developed theory of the underlying system. An alternative approach is model identification from data, which is often the only viable approach for complex systems without a developed theory. In this thesis, I apply a method called Sparse Identification of Nonlinear Dynamics (SINDy, SINDy-PI), which utilizes both approaches, identification from data using at least some limited knowledge of the system. The method will be used to identify accurate nonlinear models of two canonical systems, the Lorenz system and the pendulum-cart system. The thesis also describes methods for filtering and numerical differentiation of measured signals. During the thesis, some new adjustments are made to the original sparse regression algorithm. New ways to approach the creation of the candidate function library are also made, as well as a new way to evaluate models using clustering.

Key-words: Machine learning; system identification; sparse regression; dynamical systems; numerical differentiation; spectral filtering; spectral differentiation; model selection

Contents

List of Figures

1 Introduction

When faced with the task of controlling, predicting or understanding a given system, one of the first steps is to create a mathematical model describing it. The "traditional" approach to model creation relies on a prior established theory - a set of first principles describing the system. In practice, many systems lack such a theory, often due to high underlying complexity of the system. In these cases, we have to take a step back, and create the model from the *observations*, rather than the *principles*. The task of creating a model (discovering the principles) from observations, is known generally as *machine learning*. Many machine learning methods for discovering models of dynamical systems suffer from high model complexity, which leads into poor generalizability and poor interpretability of the model. There's an increasing interest in the area of *physically-informed machine learning*, which combines some expert system-specific knowledge with machine learning to constraint the space of possible models to the models that, for example, comply to some chosen physical laws, or are in a form that's simple and interpretable.
The method I work with in this thesis is called Sparse Identification of Nonlinear Dynamics [1], or SINDy for short. The method sets up the modeling phase as a regression task. A *sparse regression* algorithm is used to pick a subset of candidate functions from the expert-created candidate function library, so that the model is consistent with the data while being as simple ($\approx$ sparse) as possible. This methods is suitable for finding mathematical models of "theoretically elusive" systems. These systems are difficult (or impossible) to model using purely the traditional first-principle approach, usually due to high complexity. Examples of such systems can be found in disciplines like epidemiology, biology, neurology, finance, fluid turbulence, or climate. Due to time and resource constraints, I am not able to demonstrate this method on these systems; I will however show that this method is capable of identifying the cart-pendulum system with external forcing from both simulated and measurement data.

1.1 Dynamical systems

Dynamical systems are systems of equations that describe how a certain process changes in time. *Time* plays a central role in the evolution of human civilization, dynamical systems therefore have applications in most - if not all - disciplines. Dynamical systems can describe nearly everything, from the movement of stars and planets, the long and short timescale changes in the Earth's atmosphere, evolutionary dynamics, the fluctuations in capital markets to the the inner workings of our brains. Real-world processes are however *all* extremely complex and the chain of questions we have to ask to fully understand and describe them is infinite.

1.2 Mathematical models

Mathematical models of dynamical systems serve as a finite-dimensional idealization of infinitely complex processes. Imagine, for example, a *simple* oscillator - mass on

a spring connected to the ceiling. The well-known mathematical model trying to describe the dynamics of this system is fairly simple, containing only a few terms for the spring force, damping and inertial force. That is often enough to accurately describe the trajectory of the mass, given its initial state; its position and velocity. The accuracy of any mathematical model generally stands on many assumptions. The assumption that guarantees the validity of any model is that the system has no other unmodeled interactions with its environment. For an extreme thought experiment, imagine a person that's having a bad day and decides to unleash their frustration on the suspended mass, kicking into it and therefore changing its trajectory. The state of the system changed and any mathematical model running in parallel wouldn't be able to predict it. A model that would be able to predict *that* would have to contain an accurate model of the offender's brain, an accurate model of everything the brain interacts with, and by extension a model of the entire universe as well as its entire history. Due to pesky quantum effects, even this might not be enough. While this example is obviously extreme, the main point is that some unmodeled dynamics are, in practice, unavoidable. Mathematical modelling is therefore a task of describing an isolated subset of the universe. The isolation is not merely in space and time, as the task generally reduces to modeling an *idea* of the process, rather than physical reality directly.
Despite this inherent infinite complexity of real processes, the information required to describe the most important features of a process is usually finite. When modelling the oscillating mass, we can create a very precise model using only the Newton's second law and the concepts of spring and damping force. This is a case of *analytical* modeling, where we make use of a well-developed theory, the *first principles*, describing the underlying process. This theory was created by humans who observed the process, were able to identify important features and patterns and were able to describe them mathematically.
However, many real-life processes lack a theory that's as well-developed as the laws of classical mechanics. In these cases, we must do the observations, feature detection, pattern recognition and mathematical description ourselves. We have tools that can, at least to some extent, help us with these tasks. Various sensors can help us with the observations and diverse mathematical constructs and computational machines with the rest. Often the hardest task is picking the right tools and using them properly.

1.3 Law of parsimony

The best computational machines available to us today our brains. As is the case with any tool, they are often used improperly. Philosophy offers a number of heuristics, called *philosophical razors*, that can help guide us towards reasonable actions. The most well-known and useful one is *Occam's razor*, also known as the *principle of parsimony*. It tells us that when choosing between different hypotheses (models), we should value both their accuracy and simplicity. This implies that when we have two hypotheses that are equally good at describing the relevant observations, but one hypothesis is simpler than the other, the simpler explanation is likely the correct one. The definition of *parsimony*, according to Merriam-Webster [2], is "the quality of being careful with money or resources". In the context of mathematical modeling, the resource is the model complexity. High model complexity often comes with disadvantages, most notably poor model interpretability and weak ability to generalize (overfitting). The advantage of complex models is usually higher accuracy, but this also comes with some limitations. In summary, the main idea of the principle of parsimony is to ask the question *"Is the extra accuracy worth the extra complexity?"*. An optimal model will be just as complex as is necessary to accurately describe the observations.

2 State of the art

2.1 Sparse Identification of Nonlinear Dynamics

Sparse Identification of Nonlinear Dynamics (SINDy) was developed with the goal of finding simple, interpretable models from measurement data. SINDy was first introduced in the 2016 paper by S. Brunton, J. Proctor, J. N. Kutz [1]. The main idea of the method is to create a data matrix, called the function library $\mathbf{\Theta}$, which contains various nonlinear candidate functions of the state $\mathbf{x}$. Sparse regression is then used to find the vector of parameters $\boldsymbol{\xi}$ that connects the candidate functions with the target variables - the state derivatives. The method was initially only applicable for identification of dynamical systems whose underlying ordinary differential equations (ODEs) can be decomposed as linear combinations of nonlinear functions and was only able to identify natural dynamics, not the effects of external forcing. The original formulation of the SINDy method was shown to be able to exactly reconstruct the canonical Lorenz system from simulated measurement data, or find a simple model for non-stationary fluid vortex shedding behind a cylinder.

2.2 Sparse Identification of Nonlinear Dynamics with control

An extension called SINDYc [3] was developed by the original authors shortly after, allowing identification of both natural dynamics and effects of external inputs simultaneously. The trick that allows this is simple, it relies on treating input variables similarly as the state variables. In the paper, SINDYc was used to find a model of the canonical Lotka-Volterra predator-prey system with additional external inputs from simulated data.

2.3 Implicit Sparse Identification of Nonlinear Dynamics

The original SINDy method had no way to identify models of systems described by rational ODEs. Such systems are quite frequent in biology, so another extension called Implicit-SINDy was developed for this purpose by N. M. Mangan et al. in the paper called "Inferring biological networks by sparse identification of nonlinear dynamics" [4]. This method, as the name suggests, deals with rational functions by discovering implicit models rather than explicit ones. Implicit-SINDy was able to infer the models of three canonical biological systems: "Michaelis-Menten enzyme kinetics, the regulatory network for competence in bacteria, and the metabolic network for yeast glycolysis", again from simulated data. While this method worked great on clean, simulated data, it was highly sensitive to noise in the measurements. Such noise is unavoidable in real conditions, so a more robust method for identifying implicit models must've been developed.

2.4 Parallel Implicit Sparse Identification of Nonlinear Dynamics

The extension that solved the high sensitivity to measurement noise for implicit modeling was developed in 2020 by K. Kaheman et al. [5]. The algorithm, called SINDy-PI (for parallel implicit), has comparatively high time-complexity, but as the name suggests, this downside can be compensated by its parallelizability. The method introduced a trick that allows finding solutions to homogeneous problems $\mathbf{Ax} = \mathbf{0}$ using linear regression. It relies on extracting a column from $\mathbf{A}$ and putting it on the right-hand side of the equation as the target variable. The method was able to infer accurate models from many simulated systems, such as the double pendulum system or the PDE-described Belousov Zhabotinsky reaction.

2.5 SINDy for Model Predictive Control

Model Predictive Control (MPC) relies on accurate models of the controlled process. The structure of these models can be arbitrary, as long as the model is accurate. MPC is commonly used with autoregressive models, linear state-space models or neural network (NN) models. The structure of these models however doesn't necessarily incorporate any prior knowledge about the system, so the models are hard to interpret, prone to overfitting and they require a lot of training data. SINDy modeling offers an alternative that doesn't suffer as much from these issues. The combination of MPC with SINDy models was first formally suggested in the paper [6] by E. Kaiser. The paper compares many different model structures and evaluates each structure's strengths and weaknesses. Compared to NN models, SINDy models are far easier (quicker) to train, require less training data, they're easier to evaluate and more robust to noise. This makes them better suited for online identification, where the model of the process is continuously re-identified during the control system's operation to deal with changing process parameters.

2.6 Simultaneous dynamics and coordinate system identification

The SINDy method works with the assumption that the system dynamics are sparse ("simple") in some inherent coordinate system. For some processes, the inherent coordinate systems are apparent. A good example of such a process is the dynamics of a simple pendulum. A bad coordinate system would be the Cartesian coordinate system. Let's say that we track the x-y coordinate of each "particle" of the pendulum, for example using a video camera. In this coordinate system, it'd be impossible to find a simple model. A Cartesian coordinate system is not inherent to the process dynamics. If every particle was considered separately, the model would also have to be very high-dimensional. The pendulum dynamics can be expressed simply, if we assume that the pendulum is a rigid body, and if we use a polar coordinate system.
In many cases where data-driven identification is needed, the inherent coordinate system is not apparent. The measurement data are often high-dimensional. High-dimensional phenomena can be often be expressed simply if we choose a good coordinate system. The task of finding these reduced coordinate systems is known

generally as *reduced order modeling*. One approach to model order reduction would be computing the principal components (modes) using singular value decomposition and keeping only the most prominent modes. However, this offline approach might not yield coordinate systems that are good for expressing the dynamics.

A better way to find coordinate systems suitable for SINDy models was developed by K. Champion in the paper [7]. The method combines auto-encoders (a neural network architecture for dimensionality reduction) and the SINDy method for finding good coordinate systems and identifying the dynamics simultaneously. It does this by combining the cost function of the auto-encoder and the cost function of the SINDy method into one big cost function and then optimizing both auto-encoder and SINDy parameters simultaneously. The method was able to accurately identify the dynamics from a simulated Lorenz system projected into high-dimensional space, a simulated reaction-diffusion partial differential equation (PDE) model, and from a simulated video footage of a simple pendulum.

3 Identification method formulation

3.1 Dynamical systems in matrix-vector formulation

3.1.1 State-space representation of dynamical systems

Dynamical systems are described by a system of ordinary differential equations (ODEs). Systems of ODEs are most generally described by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t) + \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t). \tag{1}$$

The vector $\mathbf{x}(t) \in \mathbb{R}^n$, where n is the dimension of the state space, defines the state of the system at time t. The vector function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ describes the effect of natural dynamics at time t. The $\mathbf{u}(t) \in \mathbb{R}^b$ is the external input vector, where b is the number of external inputs, and $\mathbf{g} : \mathbb{R}^{n+b} \to \mathbb{R}^n$ is the vector function describing the effect of external forcing on the system. Note that $\mathbf{g}$ is a function of both state and input, as an input might have different effect based on the state $\mathbf{x}$. The state variables $\mathbf{x}$ are in some sense the *inner* variables, meaning they describe the full inner state of the system and not merely the desired outputs [8]. To simplify notation, I'll embed the input function $\mathbf{g}$ in the function $\mathbf{f}$, which thus becomes a function of both the state $\mathbf{x}$ and the input $\mathbf{u}$.
We express each state variable $x_i(t)$ by a separate ODE. Each state derivative variable $\dot{x}_i$ is described by a first order ODE as

$$\dot{x}_i(t) = f_i(t, \mathbf{x}(t), \mathbf{u}(t)). \tag{2}$$

If f_i is a function of time t, the dynamics themselves change with time. In some sense, the time t would in this case behave like a state variable. For most systems, we can assume that f_i isn't a function of time, and the dynamics are thus *time-independent*.
The state vector derivative $\dot{\mathbf{x}}(t)$ is defined by

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \vdots \\ \dot{x}_n(t) \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{x}(t), \mathbf{u}(t) \\ f_2(\mathbf{x}(t), \mathbf{u}(t)) \\ \vdots \\ f_n(\mathbf{x}(t), \mathbf{u}(t)) \end{bmatrix} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)). \tag{3}$$

where $\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))$ describes the vector field that defines the system dynamics.
The state-space representation provides generalization, which lets us employ any control or estimation algorithm on the state-space model without much specific knowledge about the system itself. By extension, this makes state-space models easier to work with, because if one knows how to design a control law for one state-space model, it isn't hard to apply the same sequence of steps on any other model. If the system is nonlinear, we can locally approximate the dynamics around equilibrium points by a linear model using Jacobian linearization (first order truncated Taylor expansion) [8], obtaining a linear state-space model that can be used for control. Systems of nonlinear, first order ODEs can be readily solved by numerical integration techniques such as the Runge-Kutta methods [9].

The state vector $\mathbf{x}$ can be interpreted as a point in an n-dimensional space $\mathbb{R}^n$. The dynamics of the system are governed by a vector field described by the vector function $\mathbf{f}(\mathbf{x}, \mathbf{u})$. Each point, given by the state and external input vectors $(\mathbf{x}, \mathbf{u})$, in the vector field is associated with a state derivative vector $\dot{\mathbf{x}}$. The vector field $\mathbf{f}(\mathbf{x}, \mathbf{u})$ is therefore a mapping $\mathbf{f} : \mathbb{R}^{n+b} \to \mathbb{R}^n$, where n is the state-space dimension and b is the number of external inputs.

3.1.2 Matrix-vector analytical formulation

For the original definition of the SINDy [1] method to be directly applicable, every single ODE $f_i \in \mathbf{f} = [f_1, \ldots, f_n]^{\mathrm{T}}$ must be expressible as a linear combination of some functions of the state $\mathbf{x}$ and inputs $\mathbf{u}$

$$\dot{x}_i(t) = f_i(\mathbf{x}(t), \mathbf{u}(t)) = \xi_1 \theta_1(\mathbf{x}(t), \mathbf{u}(t)) + \ldots + \xi_m \theta_m(\mathbf{x}(t), \mathbf{u}(t)) \tag{4}$$

where ξ_i are scalar parameters and m is the number of candidate functions. Simplifying the notation so that $\theta_i(\mathbf{x}(t), \mathbf{u}(t)) = \theta_i(t)$, the equation (4) can be reformulated in matrix-vector notation as

$$\dot{x}(t) = \begin{bmatrix} \theta_1(t) & \ldots & \theta_m(t) \end{bmatrix} \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_m \end{bmatrix} = \mathbf{\Theta}(\mathbf{x}(t), \mathbf{u}(t)) \boldsymbol{\xi} \tag{5}$$

where the object $\mathbf{\Theta}(\mathbf{x}(t), \mathbf{u}(t))$ is the set of candidate functions θ_i and the vector $\boldsymbol{\xi}$ is the vector of candidate function coefficients. The modeling task is to find a sparse vector of coefficients $\boldsymbol{\xi}$ that represents $\dot{x}_i$ using functions from the function library $\mathbf{\Theta}$. The sparsity condition means that we want as few elements of ξ to be non-zero. For system identification, the candidate functions θ must be picked manually. There's no prescribed way of doing this, instead it relies on some domain-specific knowledge or heuristics. For this reason, the specific choice of candidate functions θ is discussed later in this thesis in the practical sections.

3.1.3 Matrix-vector numerical formulation

Because the continuous functions $\theta(\mathbf{x}(t), \mathbf{u}(t))$ are only an abstraction that cannot be worked with numerically, we must approximate them with function measurements $\theta_i[k](\mathbf{X}, \mathbf{U})$, where k is the time-sample index. The matrix $\mathbf{X}$ is the matrix of state measurements

$$\mathbf{X} = \begin{bmatrix} x_1[0] & x_2[0] & \ldots & x_{n-1}[0] & x_n[0] \\ x_1[1] & x_2[1] & \ldots & x_{n-1}[1] & x_n[1] \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ x_1[N-1] & x_2[N-1] & \ldots & x_{n-1}[N-1] & x_n[N-1] \end{bmatrix}, \tag{6}$$

the matrix $\mathbf{U}$ is the matrix of input measurements

$$\mathbf{U} = \begin{bmatrix} u_1[0] & u_2[0] & \dots & u_{b-1}[0] & u_b[0] \\ u_1[1] & u_2[1] & \dots & u_{b-1}[1] & u_b[1] \\ \vdots & \vdots & \dots & \vdots & \vdots \\ u_1[N-1] & u_2[N-1] & \dots & u_{b-1}[N-1] & u_b[N-1] \end{bmatrix} \tag{7}$$

and N is the number of measurement time-samples. Since $\theta_i[k]$ are vectors of function *measurements*, they're not a perfect substitute for the actual functions $\theta_i(t)$. Whether the function measurements $\theta_i[k]$ represent the underlying function $\theta_i(t)$ well doesn't depend only on the sampling frequency or the number of samples, but also on the way they've been generated. If, for example, we had a function $\theta_j(t) = x_1 \sin(x_2) + 1$, but the state variable x_2 was kept constant at $x_2[k] = 0$ during the experiment, then the set of measurements $\theta_j[k]$ would be a terrible representative of $\theta_j(t)$, because it wouldn't describe the effects of any of its variables x_1 and x_2 and it would effectively look like a simple constant.

When collecting the function measurements, it is generally a good idea to sweep as much of the state space as possible. Particularly important regions in the state space are the neighborhoods of equilibrium points (points $\mathbf{x}$ where $f(\mathbf{x}) = \dot{\mathbf{x}} = 0$). The vector fields are often very variable around these equilibria, so data from these regions is likely very informative.

The discrete numerical measurement approximation of (5) has the form

$$\dot{x}_i[k] = \begin{bmatrix} | & \dots & | \\ \theta_1[k] & \dots & \theta_m[k] \\ | & \dots & | \end{bmatrix} \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_m \end{bmatrix} = \mathbf{\Theta}(\mathbf{X}, \mathbf{U})\, \boldsymbol{\xi}. \tag{8}$$

$\mathbf{\Theta}(\mathbf{X}, \mathbf{U}) \in \mathbb{R}^{N \times m}$ is the *function library matrix*, it has m columns representing the candidate functions, and N rows representing the time samples. The columns $\theta_i[k]$ representing the candidate functions are computed from state measurements $\mathbf{X}$ and input measurements $\mathbf{U}$. The state derivative measurements $\dot{x}_i[k]$ must be computed from the state measurements $\mathbf{X}$ numerically, using for example spectral differentiation described later in Section 4 that deals with data pre-processing.

The solution vector $\boldsymbol{\xi}$ is assumed to be sparse, so that only a few candidate functions have non-zero coefficients. Sparse solutions can be found using sparsity-promoting optimization methods, some of which will be described later in this thesis.

The original SINDy paper [1] didn't mention identification of systems with external inputs $\mathbf{u}$ as in the case above. The candidate functions θ_i were purely functions of the state $\mathbf{x}$. Another paper was released shortly after, where an extension named SINDYc [3] suggested including candidate functions θ_i that are functions of the control input $\mathbf{u}$. Including these functions enables the identification of actively controlled systems. This is advantageous for two main reasons. First, it lets us identify the effects of the control inputs on the system. Second, actively forcing the system lets us sweep the state-space and generate information-rich data (state trajectories) for identification.

3.1.4 Matrix-vector formulation for rational dynamics

In the equation (4), we assumed that the state variable $\dot{x}_i(t)$ could be expressed as a linear combination of functions $\theta_i(\mathbf{x}, \mathbf{u})$. This assumption isn't true for systems described by rational ordinary differential equations, which have the form

$$\dot{x}_i(t) = f_i(\mathbf{x}, \mathbf{u}) = \frac{f_a(\mathbf{x}, \mathbf{u})}{f_b(\mathbf{x}, \mathbf{u})} \tag{9}$$

where the functions $f_a(\mathbf{x}, \mathbf{u})$ and $f_b(\mathbf{x}, \mathbf{u})$ are again expressible as linear combinations of other functions. The equation (9) can be reordered into an implicit form by multiplying both sides by $f_b(\mathbf{x}, \mathbf{u})$ and then subtracting the left-hand side of the equation as follows

$$\dot{x}_i(t)\, f_b(\mathbf{x}, \mathbf{u}) = f_a(\mathbf{x}, \mathbf{u}) \tag{10a}$$

$$\dot{x}_i(t)\, f_b(\mathbf{x}, \mathbf{u}) - f_a(\mathbf{x}, \mathbf{u}) = 0. \tag{10b}$$

The equation (10b) is, again, expressible as a linear combination of some candidate functions θ_i. These candidate functions are now slightly different, they're now also functions of $\dot{x}_i$, since the denominator functions $f_b(\mathbf{x}, \mathbf{u})$ are multiplied by the state variable derivative $\dot{x}_i(t)$. The implicit ODE for $\dot{x}_i(t)$ is described by

$$\sum_{i=1}^{m} \theta_i(\dot{x}_i, \mathbf{x}, \mathbf{u})\xi_i = 0 \tag{11}$$

which can be reformulated in matrix-vector discrete numerical measurement form as

$$\boldsymbol{\Theta}(\dot{x}_i[k], \mathbf{X}, \mathbf{U})\, \boldsymbol{\xi} = \begin{bmatrix} | & \dots & | \\ \theta_1[k] & \dots & \theta_m[k] \\ | & \dots & | \end{bmatrix} \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_m \end{bmatrix} = \mathbf{0}. \tag{12}$$

This formulation was introduced in [4], in an extension called implicit-SINDy. The solution vector $\boldsymbol{\xi}$ is found as the sparsest symbol in the null-space of the function library matrix Θ. The limiting factor of this method is its high sensitivity to noise. If the measurements contain even a miniscule amount of noise, it will cause strict positivity of all singular values of the function library matrix Θ. Consequently, the dimension of the null-space increases dramatically, making the search for an appropriate solution $\boldsymbol{\xi}$ difficult.

3.1.5 Matrix-vector formulation for rational dynamics with extraction

A recently developed extension, called SINDy-PI (Parallel Implicit) [5], reformulates the problem into a more robust form. This method relies on guessing that some candidate function represented by $\theta_i[k]$ is active in the dynamics. This function θ_i is then extracted from the function library Θ and moved to the other side of the equation, becoming the new target variable and transforming the equation (12) into

$$\boldsymbol{\Theta}_i(\dot{x}_i[k], \mathbf{X}, \mathbf{U})\, \boldsymbol{\xi}_i = \begin{bmatrix} | & \dots & | & | & \dots & | \\ \theta_1[k] & \dots & \theta_{i-1}[k] & \theta_{i+1}[k] & \dots & \theta_m[k] \\ | & \dots & | & | & \dots & | \end{bmatrix} \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_{i-1} \\ \xi_{i+1} \\ \vdots \\ \xi_m \end{bmatrix} = \theta_i[k] \tag{13}$$

where Θ_i is the function library without the column $\theta_i[k]$ and $\boldsymbol{\xi}_i$ is the vector of function coefficients without the i-th element.
Using this formulation, the solution $\boldsymbol{\xi}$ can be obtained using sparse regression, which is more robust to noise. An apparent downside of this approach is that any single guess $\theta_i[k]$ is most likely going to be incorrect - the candidate function won't be present in the real dynamics. The regression problem thus has to be solved many times using different guess candidate functions θ_i. While this method is significantly more computationally demanding, the model fitting can be done in parallel for different guesses of θ_i, effectively reducing the real calculation time.

The necessity of making many guesses means that this method generates a large number of models, and finding the *best* ones isn't always trivial. The sheer number of models necessitates automated model selection, using various criteria. Model *goodness* criteria will be described later. The number of models can be reduced by exploiting the fact that the "true" model structure is likely to appear whenever the guessed function is present in the real dynamics. An often-appearing model is therefore a good candidate.

The last problem is reconstructing the model from the coefficients. The model is defined by the state variable derivative function. In explicit SINDy, this function is directly the target variable. This makes model reconstruction easy, all that's necessary is to multiply the found coefficients with the candidate functions and sum them. In implicit methods, the state derivative function is embedded in the function library Θ, and reconstruction must be done by doing the steps (10b), (10a) and (9) in this reverse order. This is not as trivial, as it requires the use of symbolic solvers for automating the modelling process. I automated the model reconstruction using the Python library called SymPy [10].

3.2 Sparse regression

3.2.1 General description

The equation (14) represents the general regression problem

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{14}$$

The objective is to find a solution vector $\mathbf{x}$ (not to be confused with the state vector $\mathbf{x}$). Using explicit SINDy formulation and notation, the regression problem can be rewritten as

$$\mathbf{\Theta}\mathbf{\Xi} = \dot{\mathbf{x}} \tag{15}$$

where $\mathbf{\Theta} \in \mathbb{R}^{N\times m}$ is the function library, $\mathbf{\Xi} \in \mathbb{R}^{m\times d}$ are the model parameters and $\dot{\mathbf{x}} \in \mathbb{R}^{N\times d}$ is the matrix of state derivatives. The number of measurement samples is given by N, the number of candidate functions by m and the number of state variables by d. The solution matrix $\mathbf{x}$ can be solved for directly, identifying all state derivatives in one function call.
In the implicit SINDy formulation, the problem is solved for each state variable derivative separately

$$\mathbf{\Theta_i}\boldsymbol{\xi}_i = \boldsymbol{\theta_i} \tag{16}$$

where $\boldsymbol{\xi}_i \in \mathbb{R}^{d\times 1}$ is the vector of parameters $\boldsymbol{\xi}$ **without** the i-th element, and $\boldsymbol{\theta}_i \in \mathbb{R}^{N\times 1}$ is the candidate function guess. The function library matrix $\mathbf{\Theta}_i$ contains all candidate functions except the target function $\boldsymbol{\theta}_i$. The notation from (14), (15) and (16) will be used interchangeably in this subsection. In the next three sub-subsections, the notations will be used in the same order as they were introduced in here. This is not an arbitrary choice, the notation reflects the intended application.

3.2.2 Regularization

There are many algorithms for solving the regression problem. Many of them put soft constraints on the solution $\boldsymbol{x}$ by incorporating the solution vector itself in the criterion function. This is known as regularization. Following the law of parsimony, we want to find a solution $\boldsymbol{\xi}$ that is *sparse*, meaning it has as few non-zero elements as possible. The soft constraint is usually obtained by incorporating the norm of the solution to the cost function. The most well-known norm is the Euclidean norm

$$L^2(\mathbf{x}) = \sqrt{\sum_{i=0}^{m-1} x_i^2} = \left(\sum_{i=0}^{m-1} x_i^2\right)^{\frac{1}{2}} = \|\mathbf{x}\|_2 \tag{17}$$

The concept of a norm is generalized by the L^p norm, defined as

$$L^p(\mathbf{x}) = \left(\sum_{i=0}^{m-1} |x_i|^p\right)^{\frac{1}{p}} = \|\mathbf{x}\|_p \tag{18}$$

Note that the L^0 norm, according to the definition in (18), is simply the number of non-zero elements of $\mathbf{x}$. Another similarly bizarre norm is the L^∞ norm, which is the limit of L^p as $p \to \infty$. The L^∞ norm of $\mathbf{x}$ then reduces as the maximum value of $\mathbf{x}$.
While norms are used for regularization, different norms are not equally difficult to implement. The L^2 norm is the most convenient and most popular, with methods like least-squares regression being built on minimizing the L^2 norm of the residuals. Some norms, such as the L^0 (sparsity) norm, are difficult to implement directly with acceptable time-complexity of the algorithm.

3.2.3 L^1 regularized least squares regression

Among the first sparsity-promoting methods is the LASSO (Least Absolute Shrinkage and Selection Operator) method introduced in [11]. It utilizes the fact that regularization by the L^1 norm happens to also minimize the L^0 norm. It's essentially an L^1-regularized least-squares method, minimizing the criterion:

$$\|\mathbf{Ax} - \mathbf{b}\|_2^2 + \lambda_1 \|\mathbf{x}\|_1 \tag{19}$$

The first term in the cost function is the sum of squares of residuals, or the L^2 norm of the residuals squared. Minimizing this term penalizes large deviations more than the small deviations. Minimizing the sum of squares of residuals promotes accuracy.
The second term is the regularizer, which sets a soft constraint on the solution by penalizing the L^1 norm of the solution. The L^1 norm penalizes all increases in the coefficients equally. It tends to set small coefficients of the solution x to zero, thus promoting sparsity. The hyperparameter λ_1 sets the weight of the L^1 penalization. It is common to do the regression multiple times with different hyperparameter values λ and then pick the best model according to some criteria.
While LASSO regression is significantly faster than combinatorial approaches, the L^1 regularization still carries a time-complexity penalty that makes the algorithm quite slow compared to the standard least-squares algorithm .

3.2.4 Sequentially thresholded least squares regression

The sequentially thresholded least squares (STLS) algorithm used in the original SINDy [1] paper uses standard least-squares regression

$$\underset{\boldsymbol{\xi}_i}{\arg\min} \|\mathbf{\Theta_i}\boldsymbol{\xi}_i - \theta_\mathbf{i}\|^2 \tag{20}$$

and then sets all elements $x_i \in \boldsymbol{x}$ that are below a defined hyperparameter threshold λ to 0. These two steps are then repeated multiple times until the solution $\boldsymbol{x}$ no longer changes between iterations. In the ODE formulation (15), the algorithm sequentially reduces the number of considered candidate functions θ_i. On the first iteration, it works with the full function library $\mathbf{\Theta} \in \mathbb{R}^{N\times m}$ and produces a non-sparse solution $\boldsymbol{\xi}$, whose elements ξ_i which are below the threshold λ are set to 0. When an element ξ_i is set to 0, its respective candidate function θ_i must also be dropped from $\mathbf{\Theta}$, so that it's no longer considered in the next iterations. The number of candidate functions usually drops significantly after the first iteration. The condition number κ [12], representing the well-posedness of the regression problem, is defined as a ratio of the biggest and the smallest singular value

$$\kappa = \frac{\max \sigma(\mathbf{\Theta})}{\min \sigma(\mathbf{\Theta})} \tag{21}$$

The condition number κ is usually very high at the first iteration, indicating the problem is ill-posed and the solution isn't reliable. In the later iterations, as columns of $\mathbf{\Theta}$ are dropped, κ also usually decreases to acceptable values.
The STLS method produces sparse solutions $\boldsymbol{\xi}$ while preserving the numerical robustness and low time complexity of least-squares algorithms. The runtime of the algorithm is orders of magnitude lower compared to LASSO, while the results are comparable. When I tested both algorithms for system identification, the STLS algorithm actually produced better results.

3.2.5 Sequentially energy-thresholded least squares regression

The STLS algorithm sequentially drops candidate functions based on parameter values ξ_i under the assumption that low values of ξ_i indicate that the respective candidate function θ_i has a small effect on the target variable and it's not relevant in the real system dynamics. However, when working with unnormalized measurements $\theta[k]$, this isn't necessarily true. Interpreting the measurements $\theta[k]$ as signals, a high-energy signal $\theta_i[k]$ with a low coefficient ξ_i might have a stronger effect than another signal $\theta_j[k]$ with a relatively higher coefficient ξ_j. To deal with this issue, I modified the STLS algorithm so that the thresholding is done based on the signal's total implied energy, defined for every signal-generating pair (θ_i, ξ_i) as

$$E(\xi_i, \theta_i) = \sum_{k=0}^{N-1} (\xi_i\, \theta_i[k])^2. \tag{22}$$

where θ_i is the i-th candidate function measurement vector and ξ_i is its respective coefficient.
This energy E is calculated for every candidate function θ present in the function library, the highest energy

$$E_{max} = \max\{E(\xi_i, \theta_i[k])\} \tag{23}$$

then becomes a baseline from which the threshold λ is calculated as

$$\lambda = E_{max}\lambda_R \tag{24}$$

where λ_R is a hyperparameter setting the relative energy ratio between the lowest acceptable energy of a signal and the maximum energy. The candidate functions θ whose implied energies are lower than λ are then dropped from $\mathbf{\Theta}$ and their respective coefficients ξ set to 0.
Instead of calculating the candidate function energies in every iteration of the algorithm, the function library $\mathbf{\Theta}$ could also be normalized during data preprocessing, so that every candidate function has the same total energy. Using the definition (22), let's define a

discrete signal $\bar{\mathbf{y}}$ so that $E(\bar{\mathbf{y}}) = 1$, we can then say

$$\begin{aligned} E(\bar{\mathbf{y}}) = 1 = \frac{E(\mathbf{y})}{E(\mathbf{y})} = \frac{\sum_{k=0}^{N-1} y_k^2}{E(\mathbf{y})} = \sum_{k=0}^{N-1} \frac{y_k^2}{E(\mathbf{y})} = \\ = \sum_{k=0}^{N-1} \frac{\sqrt{y_k^2}}{\sqrt{E(\mathbf{y})}} = \sum_{k=0}^{N-1} \frac{y_k}{\sqrt{E(\mathbf{y})}} = \sum_{k=0}^{N-1} \bar{y}_k \end{aligned} \tag{25}$$

The equation

$$\sum_{k=0}^{N-1} \frac{y_k}{\sqrt{E(\mathbf{y})}} = \sum_{k=0}^{N-1} \bar{y}_k \tag{26}$$

is satisfied if

$$\bar{y}_k = \frac{y_k}{\sqrt{E(\mathbf{y})}} \implies \bar{\mathbf{y}} = \mathbf{y} \cdot (E(\mathbf{y}))^{\frac{-1}{2}} \tag{27}$$

The signal $\bar{\mathbf{y}}$ is an energy-normalized representation of the signal $\mathbf{y}$ so that $E(\bar{\mathbf{y}}) = 1$. Normalizing every candidate function θ_i into its energy-normalized representation $\bar{\theta}_i$ would generate the energy-normalized function library $\bar{\mathbf{\Theta}}$. The coefficients x_i would then be directly proportional to the respective candidate function's energy, so we could use coefficient thresholding to get the same results as we would get with direct energy thresholding.

3.2.6 Ridge regression with sequential thresholding

Instead of using the standard least-squares algorithm with the thresholding methods, it can beneficial to incorporate Tikhonov-regularization into the algorithm, for example using the Ridge regression method first introduced in [13]

$$\arg\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 + \alpha \|\mathbf{x}\|_2 \tag{28}$$

where α defines the regularization weight. Tikhonov regularization decreases the condition number κ, which is beneficial when dealing with poorly posed problems, which in this case occur because of sometimes highly correlated candidate functions θ. Tikhonov regularization slightly increases model bias. Since the regression problem becomes better posed (κ decreases) after the first iteration when most candidate functions are thresholded out, it's possible to use Ridge regression only on the first iteration and then use the standard unregularized least squares algorithm.

4 Data preprocessing

When identifying models of dynamical systems, we need measurements of the system's state variables and their first, or sometimes even second, derivatives. Derivatives however cannot be measured directly, so they're often calculated from the measurements using numerical differentiation methods. Numerical differentiation is, due to reasons explained later, very sensitive to noise. This necessitates another preprocessing procedure - filtering. Numerical differentiation and filtering techniques used in the practical part of the thesis will be introduced in this section.

4.1 Numerical differentiation

4.1.1 Finite differences

There are many methods for numerical differentiation. The simplest one is the method of finite differences. This method uses local information about the curvature to estimate the derivative. The simplest case would be the two-point forward step differentiation, defined for continuous functions as

$$\frac{\mathrm{d}}{\mathrm{d}t}f(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{29}$$

and for discrete functions as

$$\frac{\mathrm{d}}{\mathrm{d}t}f[k] \approx \frac{f[k+1] - f[k]}{\Delta x}. \tag{30}$$

The accuracy of finite difference methods depends greatly on the step size Δx, which in the case of temporal functions corresponds to the sampling period. Lower sampling periods increase accuracy, although they also increase sensitivity to noise. Numerical differentiation algorithms in general are very sensitive to high frequency noise. Multiple-step finite differentiation increases robustness and accuracy, but at the cost of higher algorithm complexity.

4.1.2 Spectral differentiation

While finite differentiation uses the signal's time domain interpretation, spectral differentiation [14] uses the frequency domain signal interpretation. It leverages a very important property of Fourier transforms. Applying the Fourier transform on the function $y(t)$ yields its Fourier image

$$\mathcal{F}\{y(t)\} = \hat{y}(\omega) \tag{31}$$

The signal $y(t)$ is said to be the time-domain representation of the signal y, while the signal $\hat{y}(\omega)$ is the signal's representation in the frequency (or Fourier) domain.
An important property of the Fourier transform is that

$$i\omega\hat{y}(\omega) = \mathcal{F}\{\dot{y}(t)\}. \tag{32}$$

In words, this means that the Fourier image of the derivative of a function is equal to the function's Fourier image multiplied by $i\omega$, where i is the imaginary unit and ω is the angular frequency. Using the inverse Fourier transform on the equation (32), we can obtain the time derivative $\dot{y}$

$$\mathcal{F}^{-1}\{\mathcal{F}\{\dot{y}(t)\}\} = \mathcal{F}^{-1}\{i\omega y(\omega)\} = \dot{y}(t) \tag{33}$$

Spectral differentiation uses *global* information about curvature, its accuracy is generally better than the accuracy of finite differences method. The accuracy of the calculated derivative improves with smaller sampling period. One issue with this method is the presence of Gibbs phenomena (oscillations) in places where the differentiated function is discontinuous, or, in the case of discrete functions, where the function changes rapidly. This is only a major problem at the ends of our time series, since from the transformation's perspective, the derivative jumps from $\dot{y}[N-1]$ (where N is the number of samples) back to $\dot{y}[0]$ during one step. When experimenting with the method, I found a way to deal with this problem by appending a mirror image of the signal to itself, so that $y_m = [y[0], y[1], \ldots, y[N-1], y[N-1], y[N-2], \ldots, y[0]]$. Then I proceed as usual, calculating the FFT on the extended signal y_m, doing whatever frequency domain operations are necessary, transforming the signal back into time domain and keeping only the first half of the signal. This deals with most of the Gibbs "ringing" phenomena induced by first order discontinuities at the ends of the signal.
From the equation (32), we can see that the time-derivative of $y(t)$ is the function itself in the frequency domain multiplied by $i\omega$. An important point is that the object $i\omega$ is actually a high-pass filter. Numerical differentiation thus acts as a high-pass filter.
When working with sampled data - measurements - we don't use Fourier transforms directly. Instead of the function $y(t)$, we have $\mathbf{y}[k]$, the vector of measurements. When working with discrete data, we use Discrete Fourier Transforms (DFTs). The DFT is a general label used for all algorithms that calculate Fourier transforms, one such algorithm is the famous Fast Fourier Transform (FFT). FFT is implemented in most mathematical software modules, for example in MATLAB or in Python's numpy [15] module. When our signals are measurements of some physical process, the signal itself has most of its energy in the lower frequencies, while (white) noise has the same energy at all frequencies. That means that numerical differentiation decreases the signal-to-noise ratio, which raises requirements on low sensor noise and filtering methods. This is especially problematic when the signal needs to be differentiated twice, for example when we need to estimate accelerations from position measurements.

4.1.3 Comparison

To compare the two differentiation methods, I'll use measurement data from a simulation of the pendulum-cart system. The first signal x_1 is the linear position of the cart and x_2 is the angle of the pendulum. To demonstrate the high sensitivity of numerical differentiation to high-frequency noise, a comparatively weak white noise signal with a standard deviation of 0.001 is added to the signals.

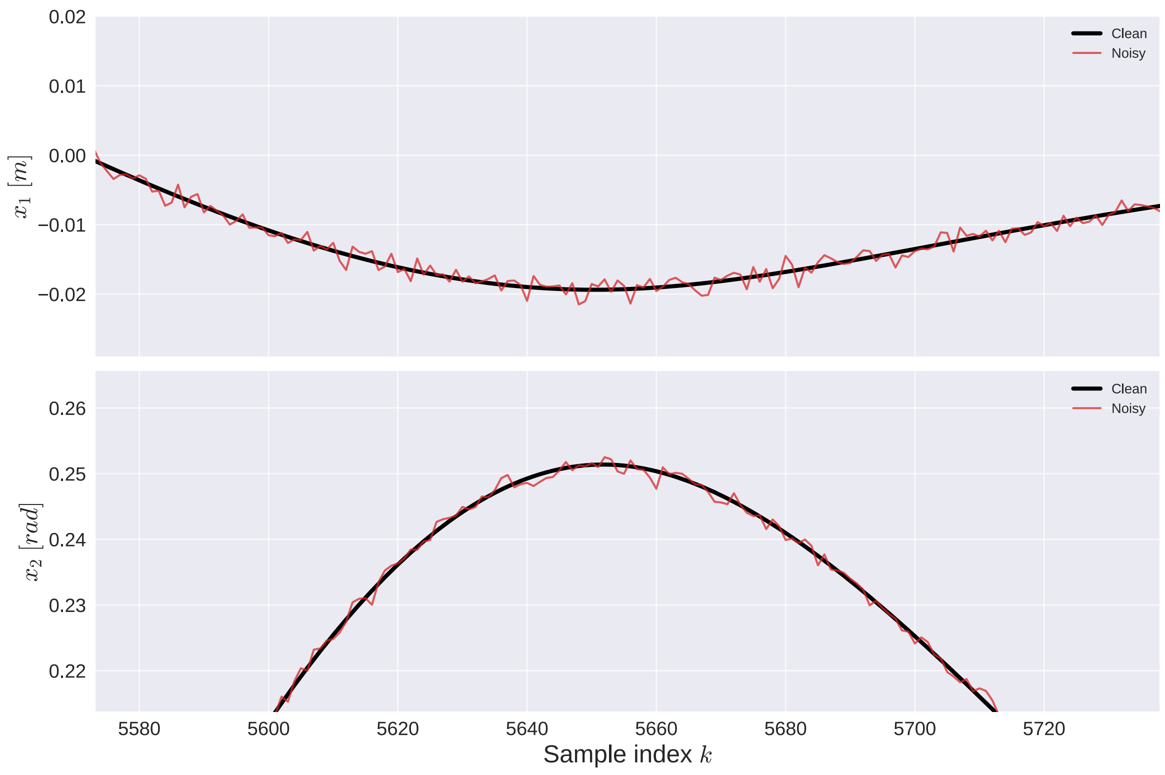


Fig. 1: Part of the signal used for demonstrating numerical differentiation algorithms. Sampling period $\Delta t = 0.001s$

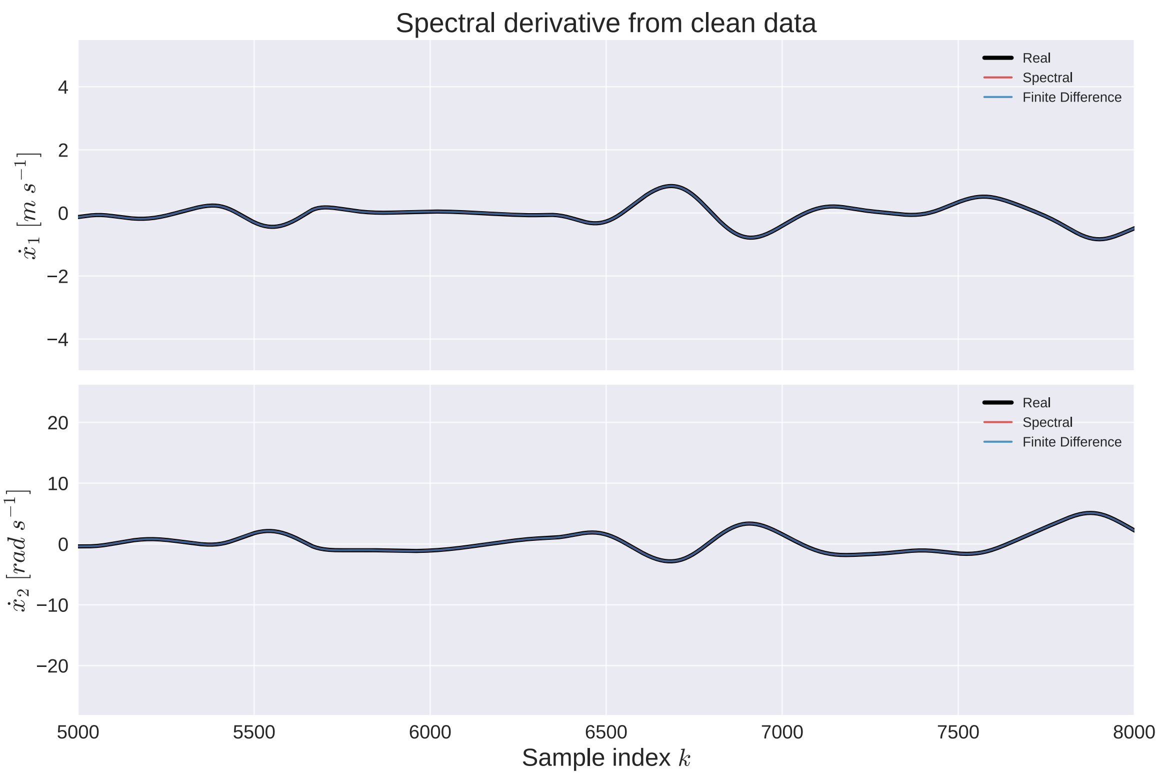


Fig. 2: Numerical derivative calculated from clean data using both algorithms. The errors are very small, so the lines are overlapping.

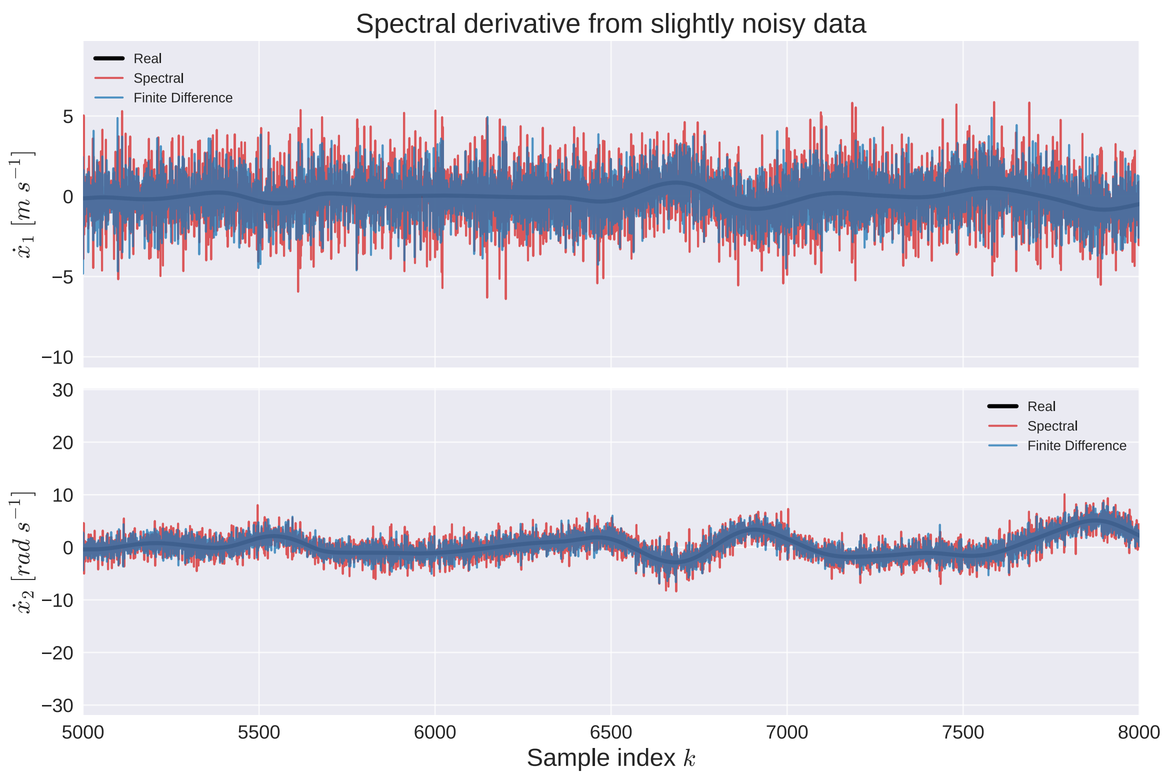


Fig. 3: Numerical derivative calculated from noisy data using both algorithms. The resulting derivative estimates are very noisy.

The simulation data used for this demonstration contains 100000 time samples with a sampling period of $\Delta t = 0.001\,\mathrm{s}$. Spectral differentiation has practically always smaller error than 2-step finite difference differentiation.
The error of the derivative estimate rises with the step size, which in the temporal case is the sampling period. The error of spectral derivative increases much slower with rising sampling period. To demonstrate this, let's down-sample the simulation data, calculate the derivatives and calculate the residual sum of squares (RSS) as an accuracy metric. RSS is defined as

$$RSS = \sum_{k=0}^{N-1} (y_{\text{real}}[k] - y_{\text{estimate}}[k])^2 \tag{34}$$

where N is the number of samples, $y_{\text{real}}[k]$ is the real derivative at time sample k and $y_{\text{estimate}}[k]$ is the calculated derivative estimate. Repeat this sequence of steps 80 times for different down-sampling periods, from the original $\Delta t = 0.001\,\mathrm{s}$ up to $\Delta t = 0.080\,\mathrm{s}$. The results of differentiation for $\Delta t = 0.080\,\mathrm{s}$ and the RSS as a function of the sampling period are shown in the next figures.

4.2 Filtering

As shown in the previous subsection, numerical differentiation behaves like a high-pass filter. This, in the presence of white noise, significantly decreases the signal to noise ratio of the resulting derivative signals. The noise in a measurement signal can be reduced either by using more advanced sensors, or by filtering the signal using either analog or digital filters.

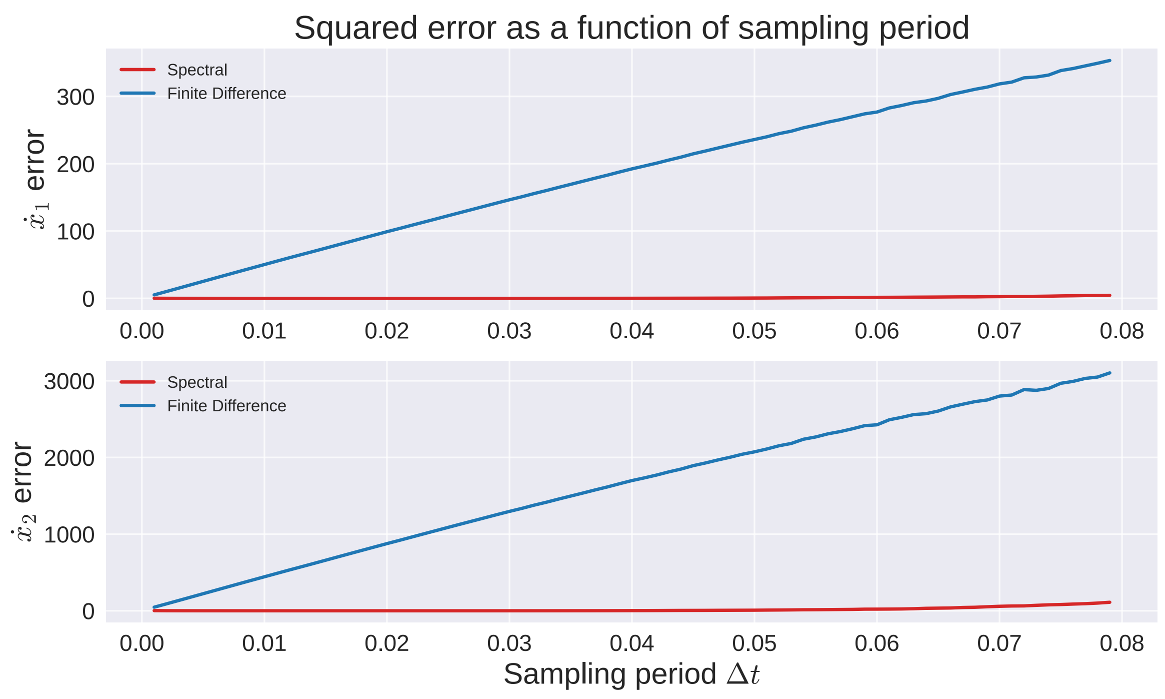


Fig. 4: Sum of squared error as a function of sampling period. The error of the spectral derivative starts lower and increases much slower.

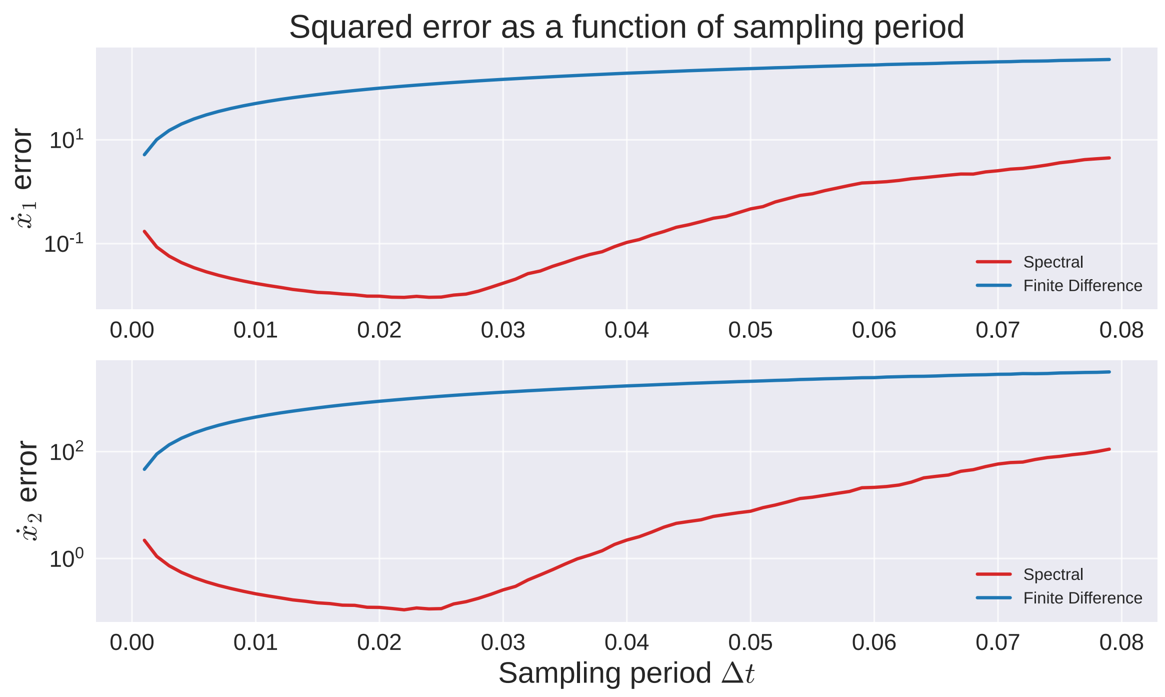


Fig. 5: Sum of squared error as a function of sampling period. Logarithmic error axis. Surprisingly, the error of the spectral derivative is actually decreasing with increasing sampling period up until about $\Delta t = 0.025s$ for both signals. I have no good explanation for this.

A filter is an object that accepts a signal as an input, separates this input based on its frequency content and outputs the desired frequency content. To properly understand filtering, one must first understand the representation of signal in time and frequency domains.

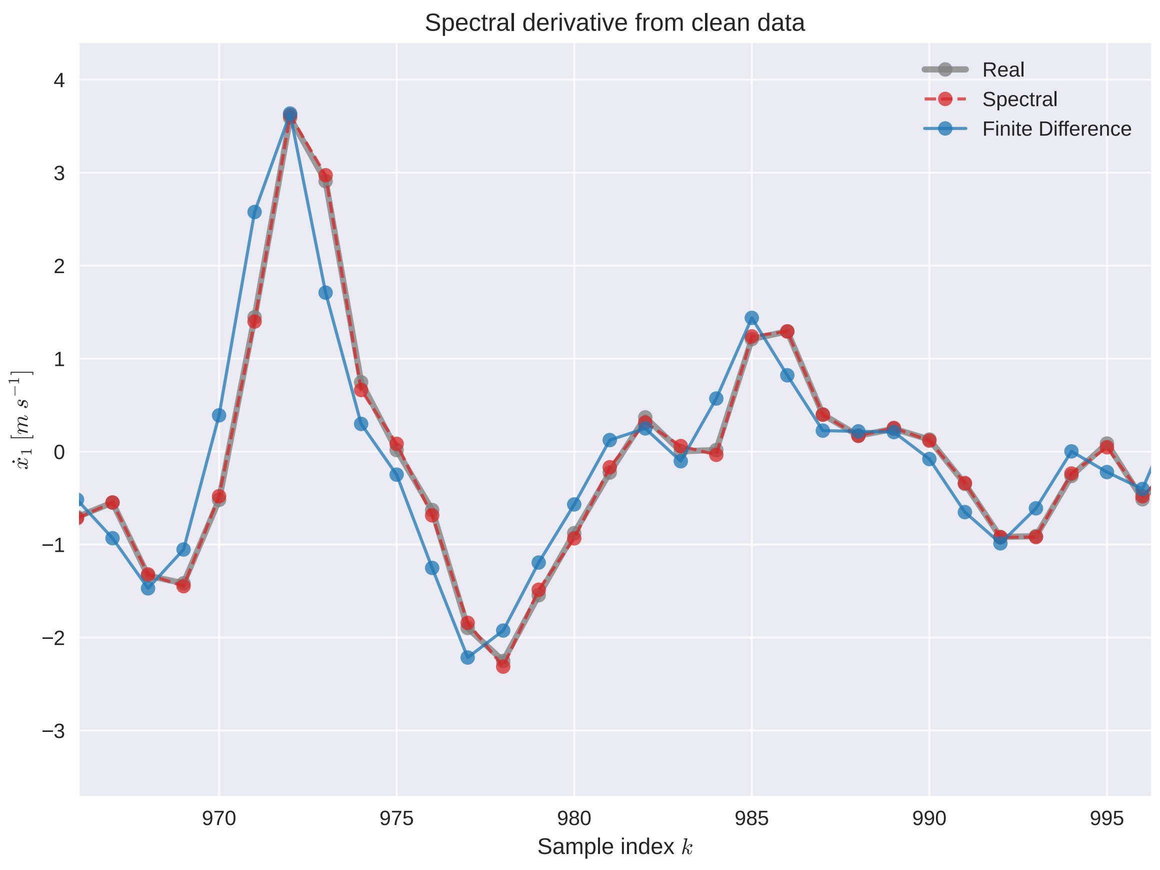


Fig. 6: Results of numerical differentiation of a downsampled signal. The signal was downsampled from $\Delta t = 0.001s$ to $\Delta t = 0.080s$. The spectral derivative estimate is nearly identical to the real derivative, while the finite difference estimate is quite off.

4.2.1 Time domain representation

Any signal can be interpreted as a sum of information and noise. Information is the part of the signal that is generated by the process we intend to actually measure. Noise is the part of the signal that's generated by other processes.
There are two ways to look at signals. We can look at them in the time-domain representation. This is the natural way for normal people to interpret signals - as a simple function of time. For a practical demonstration, I generated a random discrete information signal (signal) and a noise signal (noise). The signals are shown in Figure 7.

From now on, I'll refer to the signal+noise object as *measurement*. The measurement is simply defined as a sum of the signal and the noise. The measurement m at time sample k is

$$m[k] = s[k] + n[k] \tag{35}$$

where s is the signal and n is the noise. At each time sample k, there's only one value which stores the information. This will be a bit different in the frequency domain, making addition less trivial.

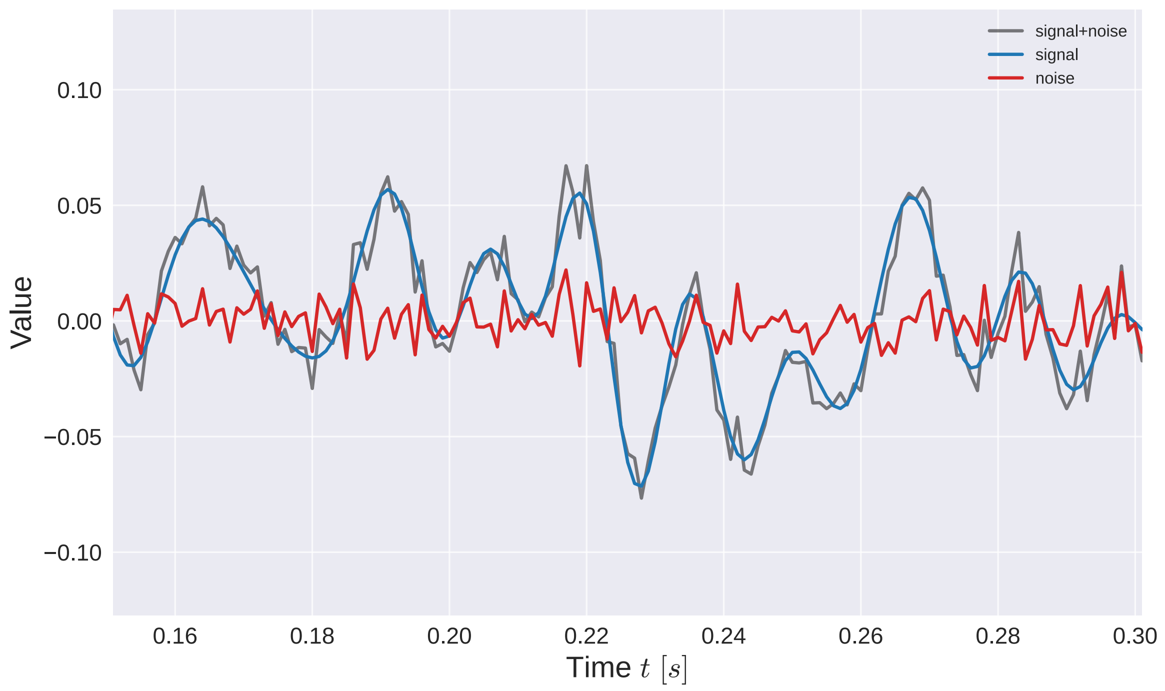


Fig. 7: A randomly generated signal and noise as a function of time.

4.2.2 Convolution filtering

An example of filtering using the time domain signal representation is convolution filtering. When we have the full input signal (typical scenario in pre-processing), we can afford to use a non-causal filter; a filter that determines the output value y at time step k by looking at input values x at time steps $l < k$ (in the past) and values at time steps $l > k$ (in the future). The value $y[k]$ is determined as a weighted sum of the values of x around time step k.

$$y[k] = \sum_{l=-(N-1)/2}^{(N-1)/2} w[l]\ x[k+l] \tag{36}$$

where w is the weighting function, also known as kernel, and N is the size of the kernel. The length of the kernel is assumed to be odd, this way the central value is strictly defined as $w[0]$. Weights $w[l < 0]$ define the weighting of input values in the past, while weights $w[l > 0]$ define the weighting of future inputs. There's uncertainty about the value of the output $y[k]$ at times $k < N$ or $k > (L - N)$ where L is the length of the time series. For these samples, the kernel is not fully overlapping with the input function. There are different ways to circumvent this problem, for example by extending the first and the last value of $x[k]$ into $k < 0$ and $k > L$ respectively.
The shape and size of the kernel defines the response of the convolution filter. Typically, the bigger the kernel, the more it suppresses higher frequencies. The values of kernel functions for de-noising typically add up to 1.
There are other approaches to computing the convolution of two signals. Utilizing the fact that convolution in the time domain is equivalent to multiplication in the frequency domain, both signals can be transformed into the frequency domain via

the Fast Fourier Transform, multiplied and transformed back into time domain. This approach is computationally efficient for longer signals and is implemented in many computational software packages, for example in Python in the *scipy.signals.convolve* [16] function.

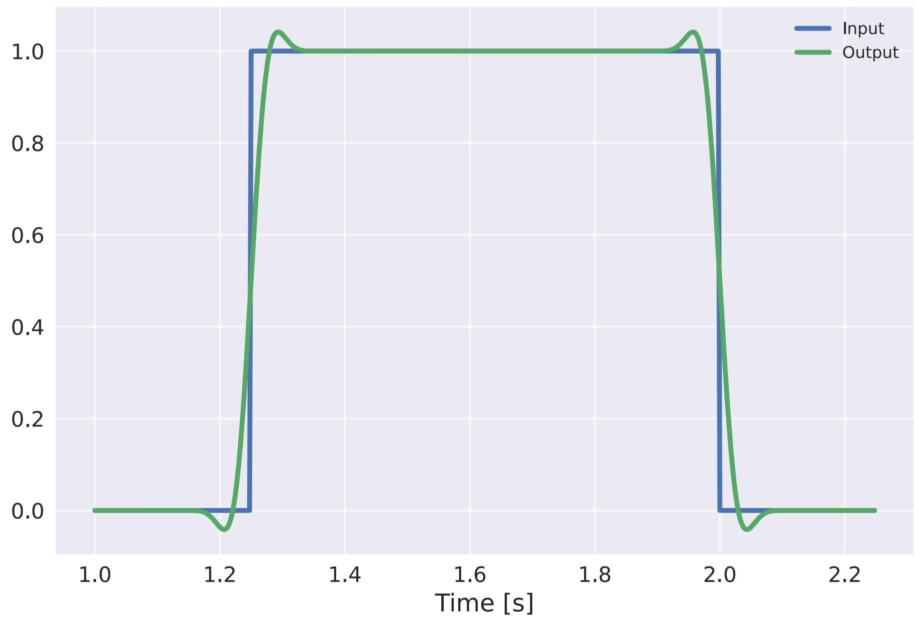


Fig. 8: Response of a non-causal convolution filter to a step input.

4.2.3 Frequency domain representation

In the frequency domain, discrete signals are defined as sums of sines and cosines with different frequencies ω, amplitudes A and phase shifts ϕ. The frequency domain's analogue to time in the time domain is, unsurprisingly, the frequency. However, while in the time domain, each time sample k was associated with only one value, in the frequency domain, each frequency ω is associated with two values - the amplitude and the phase shift. This makes signal addition more complicated. The frequency domain representation of the measurement signal is $\hat{m}$. The measurement $\hat{m}$ at frequency ω is defined by its amplitude $A_m(\omega)$ and phase shift $\phi_m(\omega)$ as a complex number

$$\hat{m}(\omega) = A_m(\omega)e^{i\phi_m(\omega)}. \tag{37}$$

According to the Euler's formula

$$e^{i\phi} = \cos(\phi) + i\sin(\phi), \tag{38}$$

the equation 37 can be rewritten as

$$\hat{m}(\omega) = A_m(\cos(\phi_m(\omega)) + i\sin(\phi_m(\omega)). \tag{39}$$

The point of this demonstration is that $\hat{m}(\omega)$ is a complex number, so summing two complex numbers $\hat{s}(\omega)$ and $\hat{n}(\omega)$ means summing their real and imaginary parts:

$$\begin{aligned}
\hat{m}(\omega) &= \hat{s}(\omega) + \hat{n}(\omega) \\
&= A_s(\omega)e^{i\phi_s(\omega)} + A_n(\omega)e^{i\phi_n(\omega)} \\
&= A_s(\omega)(\cos(\phi_s(\omega) + i\sin(\phi_s(\omega)) + A_n(\omega)(\cos(\phi_n(\omega) + i\sin(\phi_n(\omega)) \\
&= [A_s\cos(\phi_s(\omega) + A_n\cos(\phi_n(\omega)] + i[A_s\sin(\phi_s(\omega) + A_n\sin(\phi_n(\omega)] \\
&= [\mathrm{Re}(\hat{s}(\omega)) + \mathrm{Re}(\hat{n}(\omega))] + i[\mathrm{Im}(\hat{s}(\omega)) + \mathrm{Im}(\hat{n}(\omega))] \\
&= \mathrm{Re}(\hat{m}(\omega)) + i\,\mathrm{Im}(\hat{m}(\omega))
\end{aligned} \tag{40}$$

The amplitude of $\hat{m}(\omega)$ is the sum of amplitudes $A_s(\omega)$ and $A_n(\omega)$ if and only if the phases $\phi_s(\omega)$ and $\phi_n(\omega)$ are equal or different by an integer multiple of 2π. That would be a case of fully constructive addition. If the phases are in antiphase - different by $(\pi + n2\pi)$, the amplitudes subtract and the phase is equal to the phase of the stronger complex number. The addition is in this case fully destructive. If the phase difference is anywhere between these two, both the amplitude and the phase changes.
Assume we know the noise amplitude at all frequencies. Even with this information, we wouldn't be able to reconstruct the original signal's amplitude or phase information, because the noise's phase information dictates whether the addition is constructive or destructive and how the phase changes after addition.

Many processses of interest generate information signals that are mostly dispersed in the lower frequencies, or in other words, have a relatively small bandwidth. The noise, on the other hand, typically has a far bigger bandwidth. Noise is often modeled as a white noise. White noise, by definition, has infinite bandwidth and a constant power (amplitude squared) at all frequencies. Such a signal is physically not realizable. According to Parseval's theorem, such a signal would have infinite energy. In reality, the amplitude power of noise is constant up to a certain frequency, where it starts to drop off. This drop-off frequency is however typically above the sampling frequency, so it's not apparent in the measurements.

The signal shown in Figure 7 was generated as a frequency domain signal. The amplitude was defined as constant within a range of frequencies, then decreasing to 0 afterwards. The respective phases for each frequency were generated randomly. The noise signal was generated as a white noise signal, defined as having constant amplitude across all frequencies and random phase.

Note that when the (blue) signal's amplitude reaches 0, the measurement consists purely of noise. While real information signals aren't as clearly band-limited, there's always a frequency at which the noise overpowers the information. The goal of filtering is to find this (cutoff) frequency, and to preserve all frequency contents before it and discard all frequency contents after it. That way, we get rid of most of the noise while keeping most of the information.

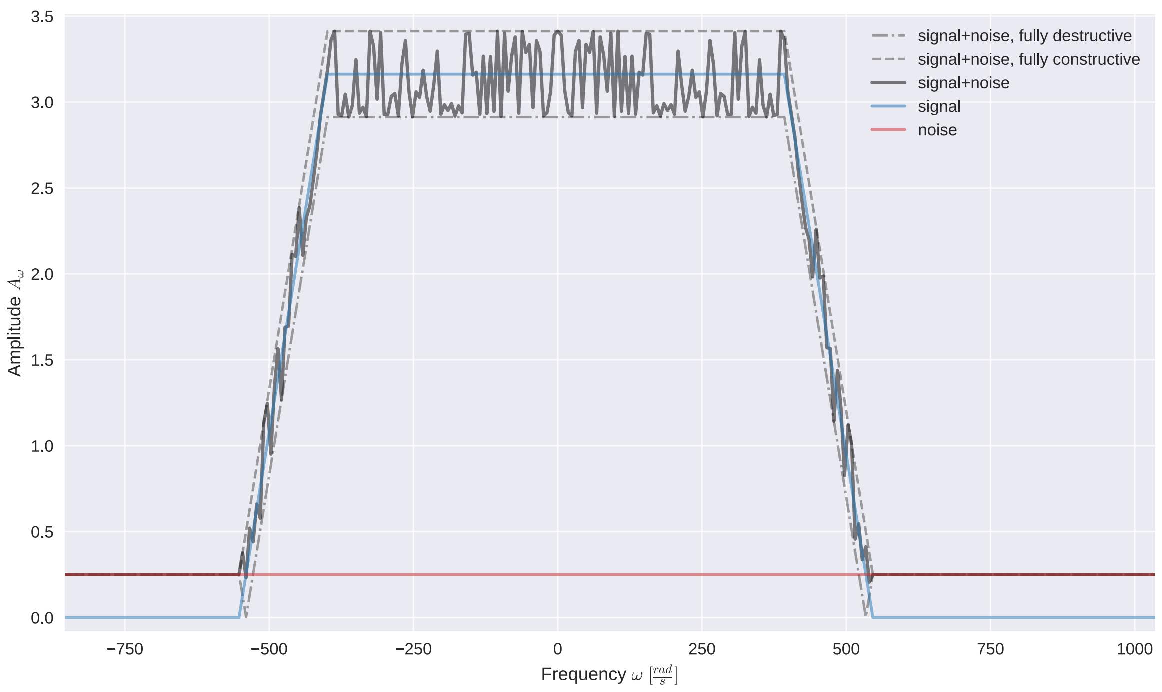


Fig. 9: The amplitude of the randomly generated signal as a function of frequency. Note the (conjugate) symmetry around 0 frequency, this is typical for purely real-valued time domain signals.

4.2.4 Spectral filtering

An ideal filter has a gain of 1 in the specified frequency range and a gain of 0 outside the range. Traditional filters cannot meet these demands. Spectral filtering techniques use the FFT to calculate the signal's representation in the frequency domain. When we have this representation, we can simply set all the high-frequency coefficients to 0 and then use inverse-FFT to reconstruct the signal in the time domain.
When designing filters, the most important design choice is the cutoff frequency. If it's set too low, then we filter out useful information from the signal, but if it's set too high, we don't get rid of the noise. In this thesis, I developed a method that chooses the cutoff frequency based on the signal's periodogram, which is an estimate of the power spectrum density (PSD).
The method works on the assumption that the signal's power spectrum density is shaped similarly as in the Figure 9, meaning the information is dispersed in the lower frequencies and then eventually drops off to 0, where the measurements contain purely noise. I choose the cutoff frequency from the periodogram, assuming the noise component of the signal is distributed evenly in the power spectrum.
I calculate the periodogram using Welch's method [17], which computes a less noisy periodogram at the cost of lower frequency resolution. The resulting periodogram is then further de-noised using a moving average filter of size 5. Moving average filters are convolution filters with a constant-valued kernel. Then I separate the higher frequency half of the signal and calculate its mean power m and the standard deviation σ. A threshold value is calculated as $m + k\sigma$, where k is a parameter defined by default as $k = 2$. The cutoff frequency is then chosen as the lowest frequency at which the

smoothed periodogram gets below this threshold value. The smoothed periodogram, threshold and chosen cutoff frequency for the generated signal are shown in Figure 10.

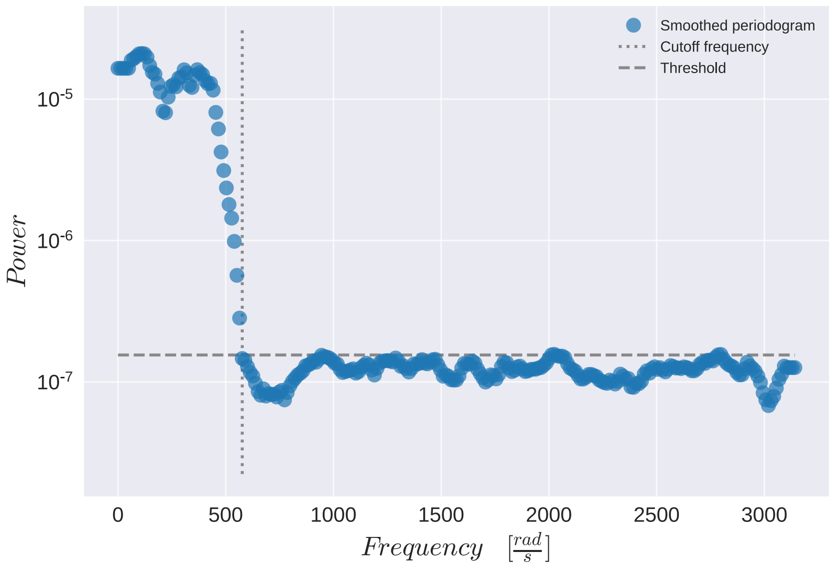


Fig. 10: The smoothed Welch periodogram, threshold value and chosen cutoff frequency. The Welch method estimates the power spectrum density with lower frequency resolution, but less noise.

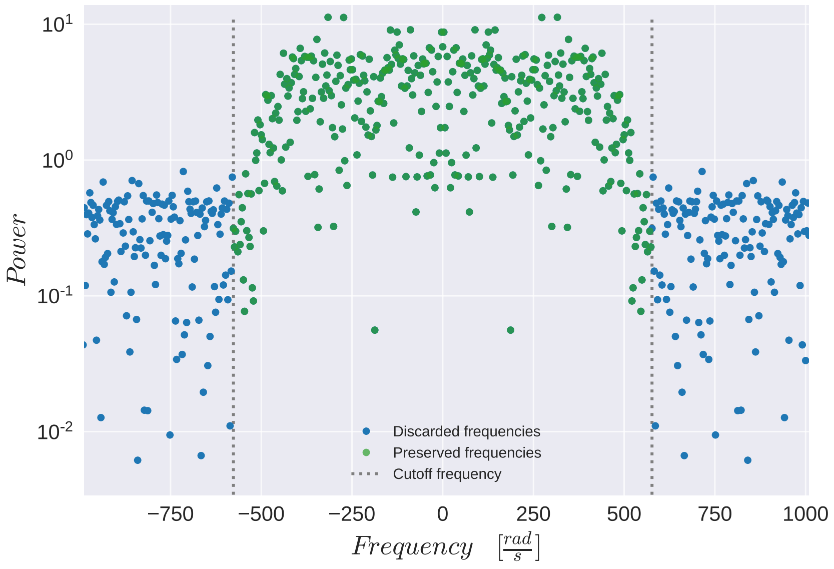
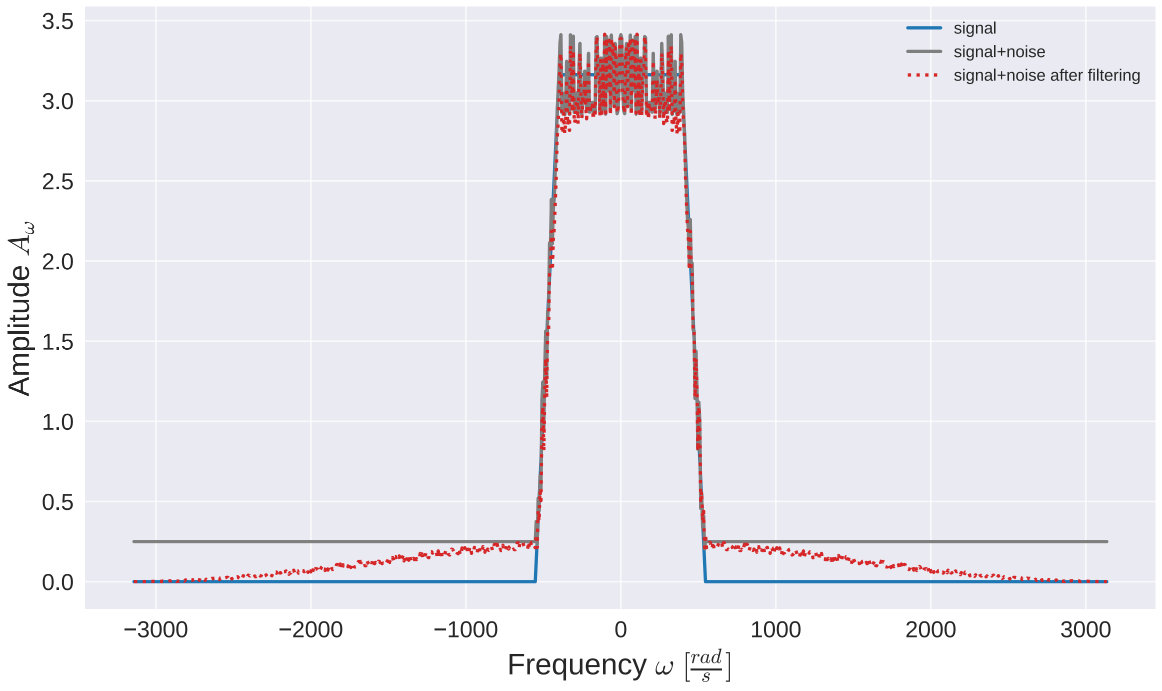


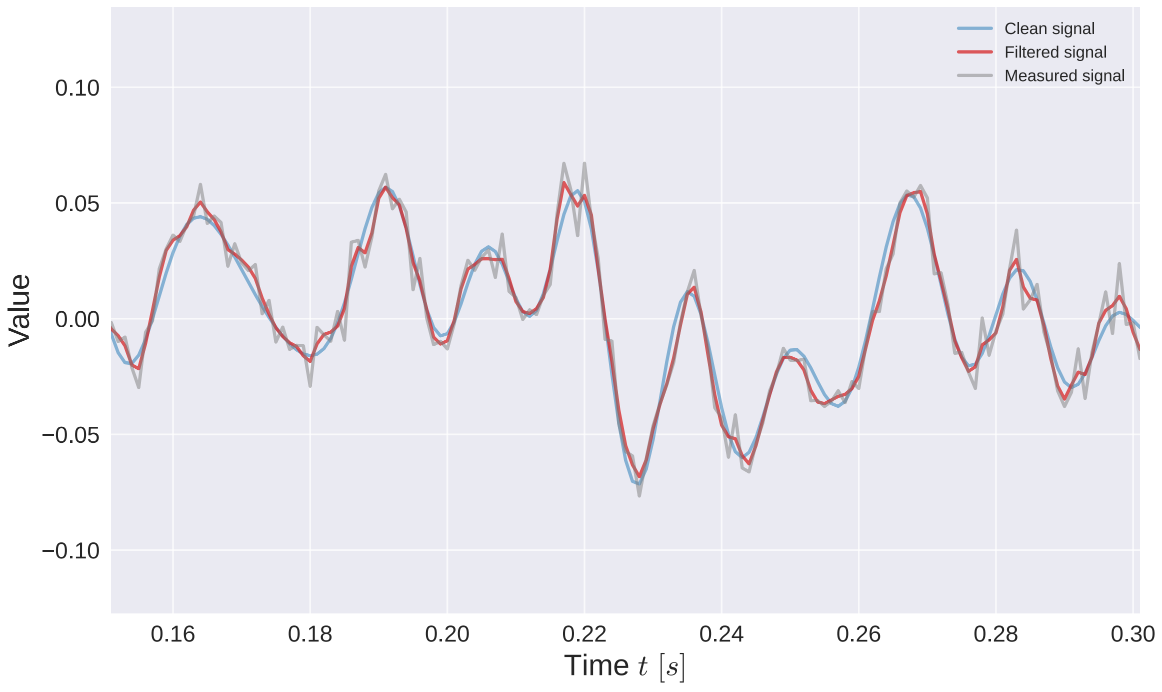
Fig. 11: The full resolution periodogram. The frequency components in the frequency band defined by the cutoff frequency are kept, components outside the band are set to 0.

4.2.5 Filter comparison

Let's use the generated noisy signal to compare different filters. Two convolution filters will be shown, both using a Hann kernel, one with a size 5 and the other with a size 9. These filters will be compared to the spectral filter, defined by its cutoff frequency chosen in Figure 10.

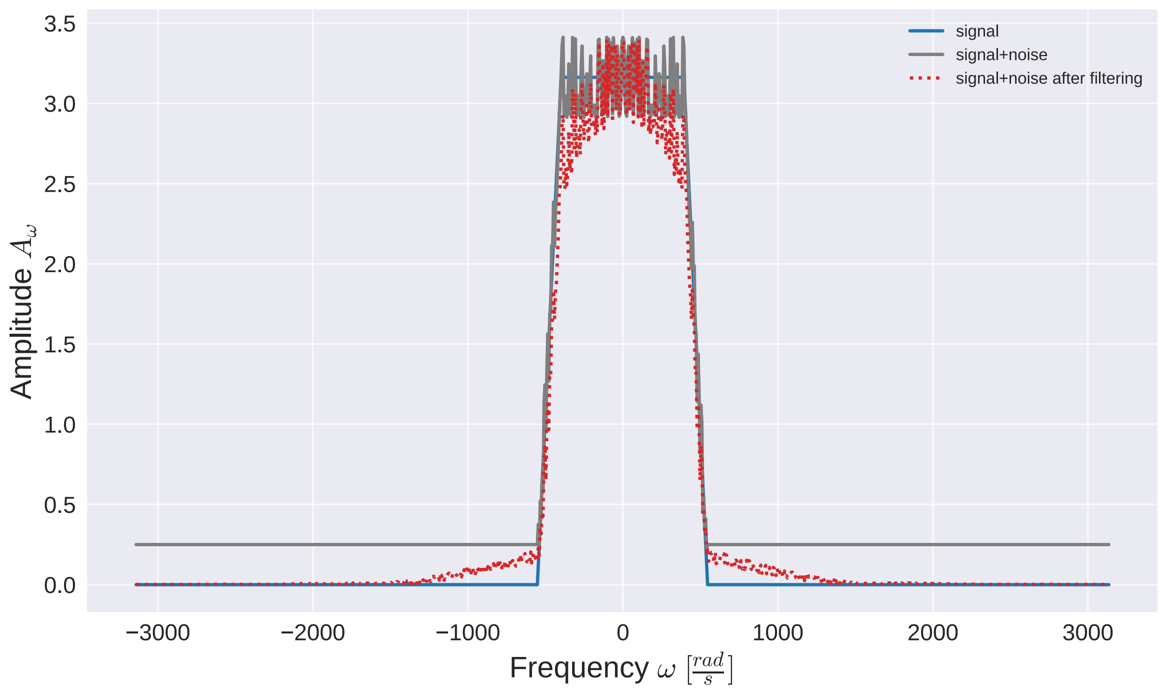


(a) The amplitude plot

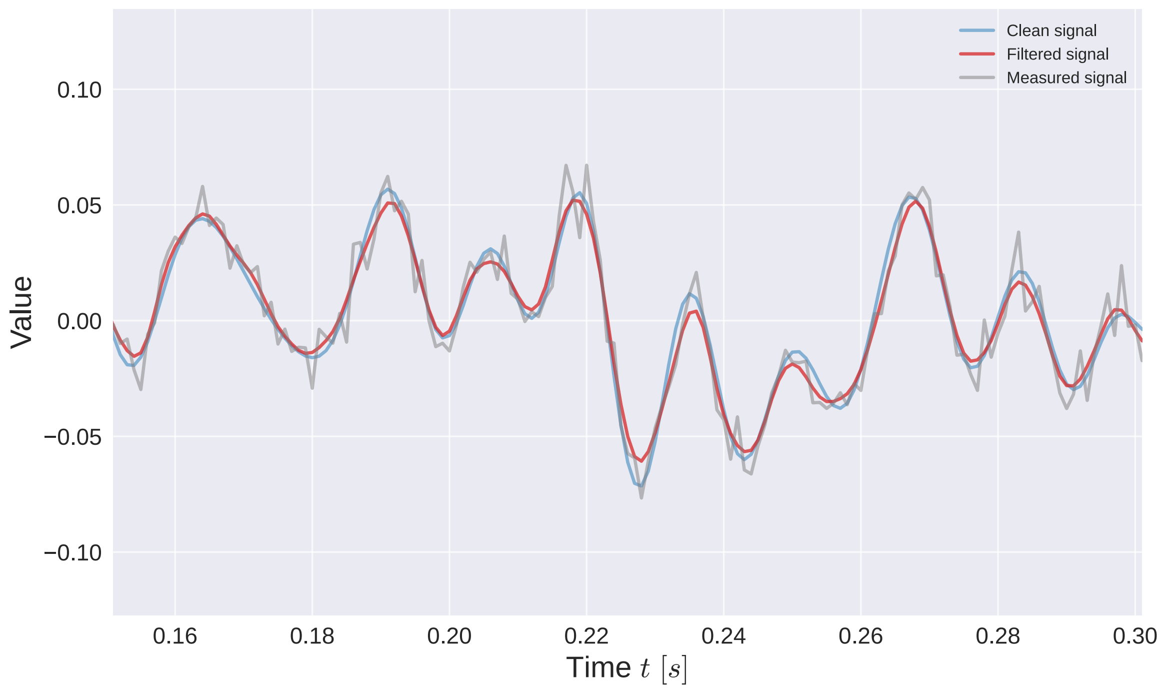


(b) The time plot

Fig. 12: Results from a Hann 5 convolution filter. Most of the information signal's spectra is preserved, however, a significant amount of noise also leaks through. In the time domain plot, the estimate follows the real derivative well. Due to the noise leak apparent from the amplitude plot, the estimate often gets thrown off by noise and produces a displeasingly jagged signal.

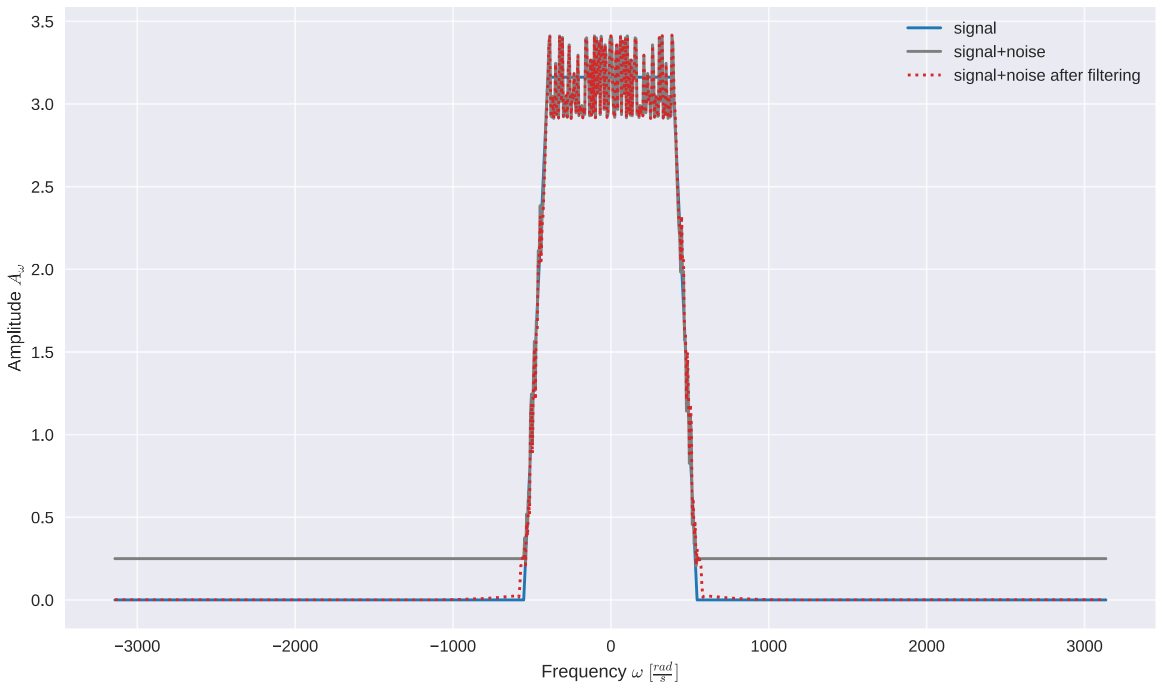


(a) The amplitude plot

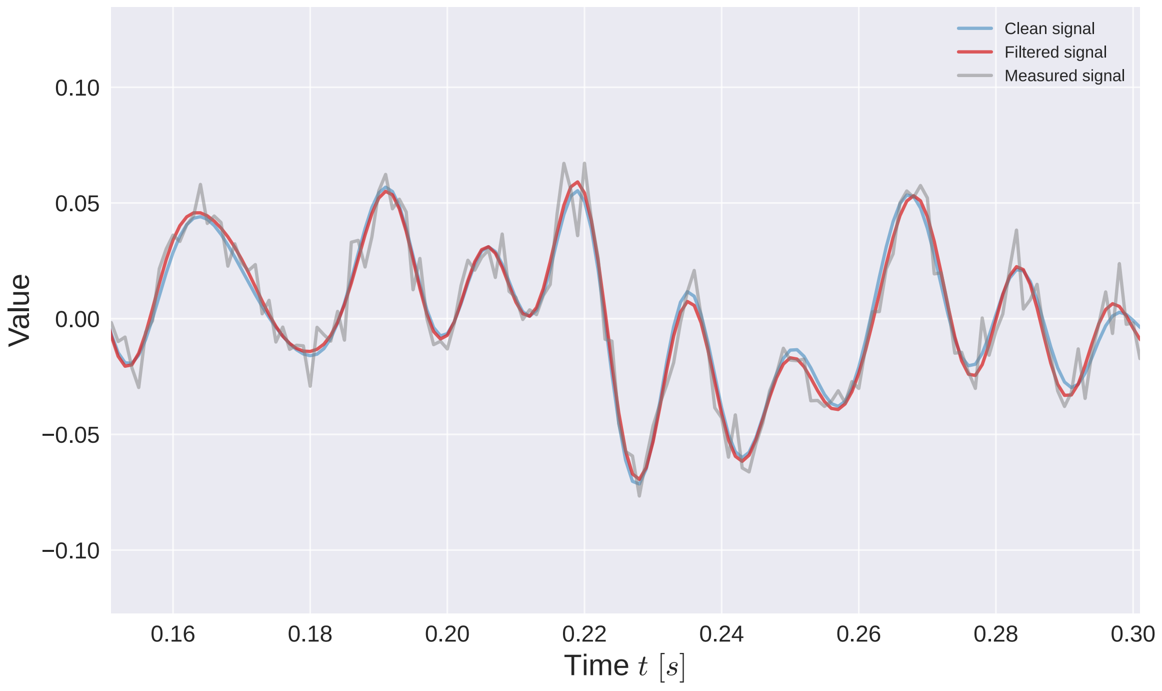


(b) The time plot

Fig. 13: Results from a Hann 9 convolution filter. A significant portion of the information signal's spectra is attenuated, which leads into a loss of information. While the signal in the time domain contains little noise, it deviates from the real derivative whenever it changes quickly. This "slowness" is a direct consequence of losing some high frequency information components of the signal.



(a) The amplitude plot



(b) The time plot

Fig. 14: Results from the spectral filter. All the frequency components after the cutoff frequency got nullified. Because the cutoff frequency was identified very accurately and the information signal is strongly band-limited, almost none of the information was discarded by the filter. Note that some information was still lost during noise addition. Reversing this corruption would require noise subtraction, which would require knowing exactly the phases of the noise frequency components.

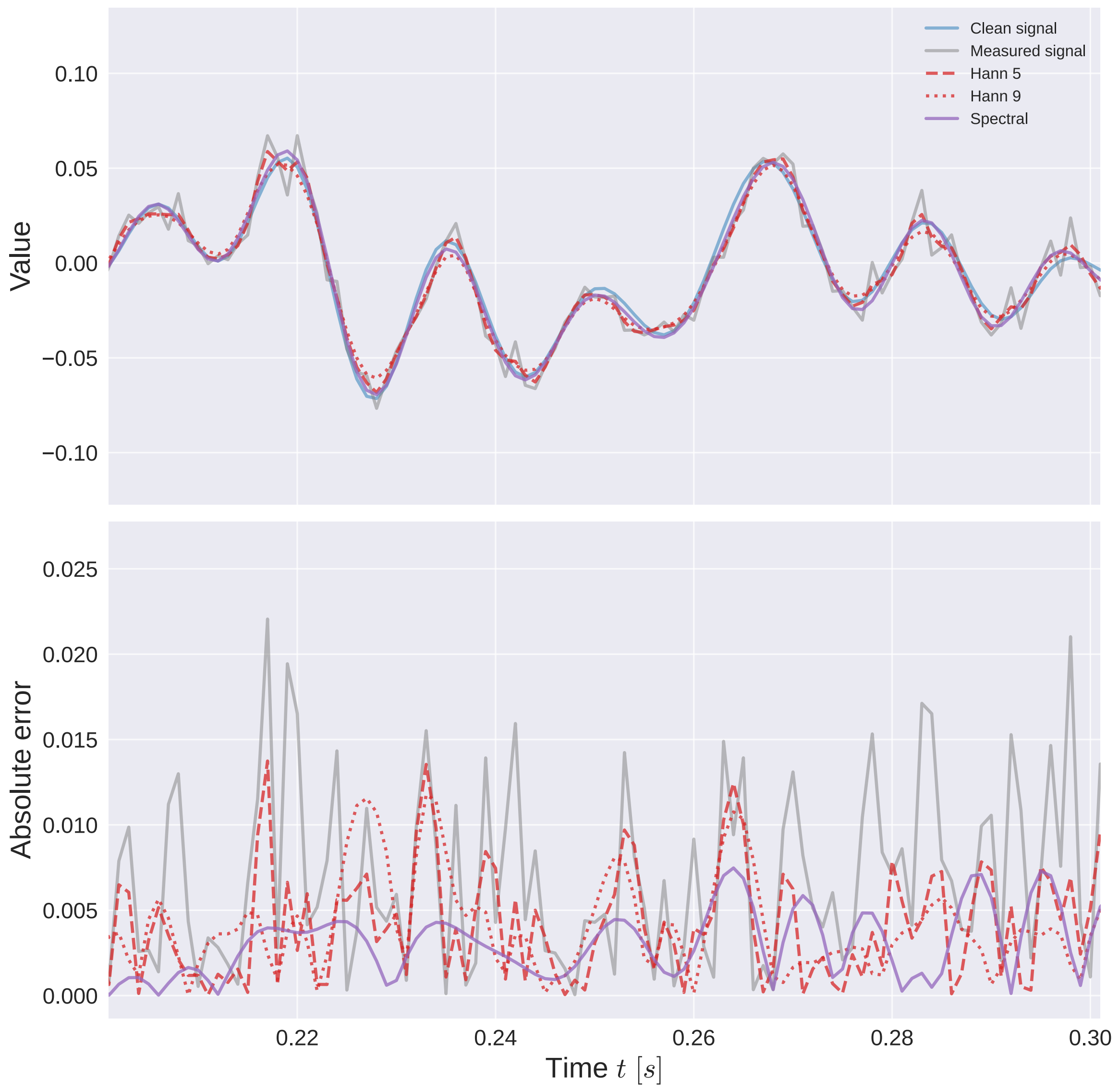


Fig. 15: Comparison of all the filters and their absolute error as a function of time.

5 Model validation and selection

5.1 Cross-validation

Because the SINDy-PI method generates a lot of models, we need a way to automate the model evaluation and selection process. To quantitatively evaluate models, we need to introduce model fit statistics. The objective of modeling is to find a model that's able to generalize to data that wasn't seen during the training (regression) phase. When creating the models, the data sets of measurements are used to create a function library $\mathbf{\Theta}$, which is then used during regression to find the solution $\boldsymbol{\xi}$ to the implicit matrix-vector equation $\mathbf{\Theta}\boldsymbol{\xi} = 0$. This estimate solution $\boldsymbol{\xi}$ is supposed to minimize the least-squares error $\|\mathbf{\Theta}\boldsymbol{\xi} - \mathbf{0}\|^2$ while also minimizing the number of non-zero parameters $\boldsymbol{\xi}_i$, or in other words, the L^0 norm of $\boldsymbol{\xi}$. Because this function library is used to *train* the model, it will be referred to as the *training* set. We're interested in models that are able to generalize, so we need to evaluate the models on a different data set, called the *test* set.
The test set is a set of state and input measurements that wasn't used for training the model. When we're simulating the system, it can be generated by a different simulation. The simpler way is to split the original measurements into a training set and a test set before constructing the candidate function library. Note that when the function library is generated, its samples (rows) can be arbitrarily shuffled - they don't have to be in chronological order, as with filtering and differentiation.

5.2 Calculating model errors

All model goodness metrics require first calculating the model errors. Errors are the difference between the real values and the values estimated by the model. While the training errors are calculated during the training phase, the validation errors are calculated after training, in the model selection phase. Practically, there are two ways to calculate them. The errors (residuals) can be calculated from $\mathbf{\Theta}$ and the solution $\boldsymbol{\xi}$ directly, assuming an implicit model, as

$$\epsilon = \mathbf{\Theta}\boldsymbol{\xi} \tag{41}$$

Note that this only holds when the "target variable" is the null vector $\mathbf{0}$. I use the equation above to calculate the training errors for each model. The result is an $N \times 1$ vector, one element for each row of the $\mathbf{\Theta}$ matrix. Calculating the error like that requires the data to have the structure of the candidate function library $\mathbf{\Theta}$. Because I generate an ODE function for each model automatically, the errors can also be calculated using the model itself. This is done by passing the state and input values from the test set into the model ODE function and subtracting the test set's "real" derivative from the result:

$$\epsilon = \dot{x}_{\mathbf{model}}(\mathbf{x}_{\text{test}}, \mathbf{u}_{\text{test}}) - \dot{x}_{\text{test}} \tag{42}$$

I use this approach to calculate the validation errors, as the validation data doesn't have to be in the form of a function library.

5.3 Model fit metrics

The model goodness metrics in this thesis are all calculated from the error vectors ϵ. For each model, we can calculate a single scalar metric indicating how well it fits the reference dataset. Whether the reference dataset is the testing or training one depends on how the error vector ϵ was calculated. A simple statistic evaluating the accuracy of the model is the mean-squared-error (MSE), which is calculated generally as

$$\mathrm{MSE}(\mathbf{\Theta}, \boldsymbol{\xi}) = \frac{1}{N} \sum_{k=0}^{N-1} \epsilon_k^2 \tag{43}$$

The physical units of MSE are the original unit squared. By taking the square root of the MSE, we get the root-mean-squared-error metric, RMSE. The unit of RMSE is identical to the target variable's unit.

5.4 Information criteria

According to the law of parsimony, we should also take into account the model's complexity. A particularly important metric, which balances a model's accuracy and its complexity, is the Akaike Information Criterion [18], or AIC for short. AIC is generally calculated as

$$\mathrm{AIC} = 2K - 2\log L(\boldsymbol{\xi}|\mathbf{\Theta}) \tag{44}$$

where $\log L(\boldsymbol{\xi}|\mathbf{\Theta})$ is the log-likelihood of parameters $\boldsymbol{\xi}$ given data $\mathbf{\Theta}$ and K is the number of terms, which in our case is the number of non-zero elements of $\boldsymbol{\xi}$. In the case of least-squares regression and the assumption that errors are normally distributed [19], AIC can be reduced to

$$\mathrm{AIC} = 2K + N\log\mathrm{MSE} \tag{45}$$

The AIC is then calculated for every identified model. The AIC scores for each model are then compared, and the models with lowest AIC are considered for further selection. The Akaike Information Criterion builds on previous concepts in information theory, particularly the Kullback-Leibler divergence [20].

In this thesis, I didn't find this criterion particularly useful. Since the models are sparse, the $2K$ term is relatively tiny compared to the $N\log\mathrm{MSE}$ term. The criterion thus effectively reduces to being a MSE metric. The criterion would be more useful if there were bigger differences in the numbers of parameters between candidate models.

6 Implementational aspects of the SINDy method

6.1 Brief recap of the SINDy-PI method

Before the practical part of the thesis, let's do a quick recap of the SINDy-PI method and describe some steps I made when implementing the identification method. The goal of SINDy is to find a sparse, nonlinear model describing the dynamics of the studied system. Sparsity in this context means that the resulting equations contain as few terms as possible. The SINDy-PI [5] extension used in the practical part of the thesis allows the identification of dynamics described by rational functions by changing the structure of the function library and tweaking the regression step of the identification. The method tries to identify **implicit** ordinary differential equations describing the dynamics. Naturally, SINDy-PI can also be used to identify non-rational dynamics. For this reason, I'll always use the SINDy-PI algorithm for identification, so that I save on implementation time. The SINDy-PI regression task is, for each separate ODE in the system, in the form

$$\mathbf{\Theta_i}(\mathbf{X}, \dot{\mathbf{X}}, \mathbf{U})\, \boldsymbol{\xi}_i = \theta_i \tag{46}$$

where $\mathbf{\Theta_i} \in \mathbb{R}^{N\times(m-1)}$ is the candidate function library with the function $\theta_i \in \mathbb{R}^{N\times 1}$ extracted out. The $\boldsymbol{\xi}_i$ is the vector of parameters of the candidate functions from $\mathbf{\Theta}_i$. To enable implicit model identification, the function library $\mathbf{\Theta}(\mathbf{X}, \dot{\mathbf{X}}, \mathbf{U})$ is built not only from state measurements $\mathbf{X}$ and input measurements $\mathbf{U}$, but also from state derivative measurements $\dot{\mathbf{X}}$. This is the main difference between implicit and explicit SINDy methods. In explicit methods, the state derivative measurements $\dot{\mathbf{X}}$ wouldn't at all be embedded in the function library $\mathbf{\Theta}$; instead they'd be on the other side of the equation acting as the target variables.

As the system identification engineers, we have full freedom to choose the specific candidate functions in $\mathbf{\Theta}$. This gives us the ability to impute system-specific knowledge into the identification process. Let's quickly demonstrate this concept on the example of data-driven model identification of a simple pendulum. Anybody who has ever derived the system's dynamics from first principles surely remembers that the equations of motion contained trigonometric terms - particularly sines and cosines. This is a piece of knowledge that's not enough to derive the model analytically, but it can be imputed into the data-driven identification process by, for example, including trigonometric functions of the pendulum's angle in the function library. The identification engineer can, to a limited extend, make mistakes when choosing these candidate functions by including functions that aren't at all present in the dynamics. The identification method rectifies these mistakes through sparse regression, where the parameters of these "wrong" candidate functions are simply set to 0 and discarded.

A more interesting example would be including friction models. The pendulum surely loses some energy during its movement to some kind of resistance. By including various friction models into the function library, the method could, in theory, identify even which friction models are active and their parameters. These friction models could be highly nonlinear, even discontinuous, as long as they can be expressed in the implicit ODEs linearly with respect to their parameters. For example, if ξ_i is the function's parameter and x_1 is a state variable, the candidate function θ_i could be $\xi_i x_1^2$, $\xi_i \mathrm{sgn}(x_1)$ or $\xi_i \mathrm{ReLU}(x_i)$, but not $\xi_i^2 x_1$ or $x_1 \sin(\xi_1)$.

6.2 Representing the problem in implicit form

After finding the solution $\boldsymbol{\xi}_i$ using sparse regression, the extracted function θ_i can be imputed back into the i-th column of $\boldsymbol{\Theta}_i$ and -1 can be imputed back into the respective row of $\boldsymbol{\xi}_i$. This way, the (46) problem is rewritten back into the implicit form

$$\boldsymbol{\Theta}\,\boldsymbol{\xi} = \mathbf{0}. \tag{47}$$

For implementational purposes, this form is very convenient, because the structure of $\boldsymbol{\Theta}$ and $\boldsymbol{\xi}$ stays constant - each unique candidate function corresponds to a unique column of $\boldsymbol{\Theta}$ and a row of $\boldsymbol{\xi}$. The parameter vectors $\boldsymbol{\xi}$ of the identified implicit models can thus be compared directly, even if they used different guess candidate functions θ_i.

6.3 Representing the implicit models

For each model, I generate a characteristic string describing the identified implicit equation. This string is constructed using the column identifiers of $\boldsymbol{\Theta}$ (the candidate function names) and their respective parameters from the parameter vector $\boldsymbol{\xi}$. Remember that in the implicit form, even rational ODEs can be expressed as a linear combination of candidate functions. The characteristic string represents this linear combination. For example, if $\boldsymbol{\Theta}$ had 5 candidate functions with the string identifiers `[x_1, dx_1, dx_1*x_1, x_2, u]` and the regression found $\boldsymbol{\xi}$ to be $[1, 0, 2, 3, 2]^{\mathrm{T}}$, the characteristic string would be `1*x_1 + 2*dx_1*x_1 + 3*x_2 + 2*u`.

This characteristic string serves many purposes. Other than being a characteristic identifier for each model, it's also passed into SymPy's[1] [10] parser to produce SymPy's representation of a symbolic function. SymPy's algebraic solver can then reorder the implicit function into its explicit ODE form. The explicit ODE symbolic equation can then be used to automatically generate an ODE function using either SymPy's `codegen` function to generate MATLAB function files, or its `lambdify` function to generate Python's anonymous functions. Automatic code generation is a massive advantage, as even relatively simple models are tedious to manually code into an ODE function for simulation purposes. SymPy enables quick and painless transition from identification to simulation, speeds up the engineer's workflow and preserves their nerves.

6.4 Reducing the number of models

The SINDy-PI method generates a very large number of models. There are m candidate functions θ in the function library $\boldsymbol{\Theta}$, if each one of them is used as a guess, each generates at least one implicit model. Each guess function generates a number of

[1]An open-source symbolic math module for Python. Stands for "Symbolic Python".

models that depends on the cardinality of the set of regression hyperparameters we specified before starting the identification. For example, if we defined the hyperparameter set as $\Lambda = \{\lambda_1, \lambda_2, \lambda_3\} = \{0.1, 0.01, 0.001\}$, then for each guess function, the regression will be performed 3 times, each time with a different hyperparameter value, and each regression will yield a model. Given that the number of candidate functions m can be in the hundreds, we need a quick way to sift through the massive number of models.

Different hyperparameter values λ can sometimes yield exactly the same models. The first (and easiest) way to reduce the number of models is therefore to only keep unique models. I create an unique fingerprint for each model by sending the model's guess function string and its parameter vector $\boldsymbol{\xi}$ into a hash function. If there are two or more models with the same hash, only the first one is preserved and the others are discarded.

The other way to reduce the number of models used in this thesis leverages the fact that the correct model should be identified for many different guess candidate functions θ_i. Therefore, models that appear consistently in the generated set of models are good candidate models. To understand consistency, let's first define the dissimilarity between two models. One way to define the dissimilarity between two models is though their parameter vectors $\boldsymbol{\xi}$. We can calculate the parametric distance between two models defined by the parameter vectors $\boldsymbol{\xi}_i$ and $\boldsymbol{\xi}_j$ respectively. Both vectors exist in $\mathbb{R}^m$, where m is the number of candidate functions θ. The difference between model j and model i can be defined as the difference between their parameter vectors $\boldsymbol{\xi}$

$$\Delta(\boldsymbol{\xi}_i, \boldsymbol{\xi}_j) = \boldsymbol{\xi}_i - \boldsymbol{\xi}_j. \tag{48}$$

Taking, for example, the L^2 (Euclidean) norm of this difference vector gives us a single number describing the dissimilarity between two models. If two models are close together in the parametric space, they're "similar" and the dissimilarity metric will be small.

For the purposes of sparse identification, I found a better definition of model consistency. The dissimilarity metric I use to find consistent models relies on transforming the parameter vectors $\boldsymbol{\xi}$, so that non-zero elements are set to 1 and the 0 elements stay 0. More formally, I calculate the **activation** vector $\mathbf{a}(\boldsymbol{\xi})$ by the element-wise operation

$$a_i = \xi_i^0 \qquad (\xi_i \text{ to the } 0\text{−th power}) \tag{49}$$

The activation vector $\mathbf{a} \in \{0,1\}^m$ then replaces the parameter vector $\boldsymbol{\xi}$ for calculating model similarity. If two models have the same active (non-zero) terms, the difference vector between their activation vectors is the null vector $\mathbf{0}$. For a scalar dissimilarity metric, I use the L^1 norm of the activation difference vector $\Delta(\mathbf{a}_i, \mathbf{a}_j)$. This literally corresponds to the number of different active terms.

To then find groups of consistent models, I use hierarchical clustering. Hierarchical clustering is computationally intensive for even moderately sized datasets, but our number of elements (models) is very small in the context of automated data analysis.

It has the advantage of being relatively simple and being able to find even "weirdly shaped" clusters. I define the cutoff activation distance as 2, meaning that if two models have a bigger activation distance bigger than 2, they'll be assigned into different clusters. The end result of clustering is a cluster label for each model. The models that are in a non-singular cluster (there's at least one other model with similar structure) are kept for further validation and the other models are discarded.

7 Model identification of the Lorenz system with feed-forward inputs

To demonstrate the method, I'll identify the model of the canonical Lorenz system. The system equations will be expanded by three external inputs to show the method's ability to identify the effects of forcing as well. The natural dynamics are fairly simple, consisting only of terms created from x_1, x_2, x_3. To show that the effect of external variables is also identifiable, I'll also add separate forcing terms u_1, u_2, u_3 to each of the differential equations.

7.1 Definition

Let's define the Lorenz system with external input as

$$\dot{x}_1 = \gamma(x_2 - x_1) + 40\mathrm{sgn}(u_1) \tag{50a}$$

$$\dot{x}_2 = x_1(\rho - x_3) - x_2 + 10\sin(u_1)u_2 \tag{50b}$$

$$\dot{x}_3 = x_1x_2 - \beta x_3 + u_3x_1 \tag{50c}$$

where $\dot{\mathbf{x}} = [\dot{x}_1, \dot{x}_2, \dot{x}_3]^{\mathrm{T}}$ is the state derivative, $\mathbf{x} = [x_1, x_2, x_3]^{\mathrm{T}}$ the state and $\mathbf{u} = [u_1, u_2, u_3]^{\mathrm{T}}$ the vector of external inputs. Note that the external inputs also enter the ODEs non-linearly. The $\mathrm{sgn}(u_1)$ function is the sign function

$$\mathrm{sgn}(u_1) = \begin{cases} 1, & \text{if } u_1 \geq 0. \\ -1, & \text{if } u_1 < 0. \end{cases} \tag{51}$$

The other parameters are chosen as $\sigma = 10$, $\beta = \frac{8}{3}$ and $\rho = 28$. By substituting the parameter values for the parameters in (50), the true system's dynamics are

$$\dot{x}_1 = 10x_2 - 10x_1 + 40\mathrm{sgn}(u_1) \tag{52a}$$

$$\dot{x}_2 = 28x_1 - x_1x_3 - x_2 + 10\sin(u_1)u_2 \tag{52b}$$

$$\dot{x}_3 = x_1x_2 - \frac{8}{3}x_3 + u_3x_1. \tag{52c}$$

To identify how the inputs $\mathbf{u}$ affect the state derivatives, the inputs must be doing something during simulation. Therefore, I'll set the inputs $\mathbf{u}$ as functions of time t. More specifically, I'll define them as a band-limited noise process. For each input, I'll generate a white noise signal with some defined power. These white noise signals will then be low-pass filtered using a spectral filter to get rid of higher frequencies. For practical purposes, it's necessary to get rid of the higher frequencies in the input signal. The measurements will also be low-pass filtered to allow for numerical differentiation, so the effects of higher frequency input signals would also get filtered out. Also if the inputs were physical actuators, sending high frequency and relatively high power signals into them might damage them or the system itself. The signal sent into the input u_1 during simulation is shown in Figure 16.

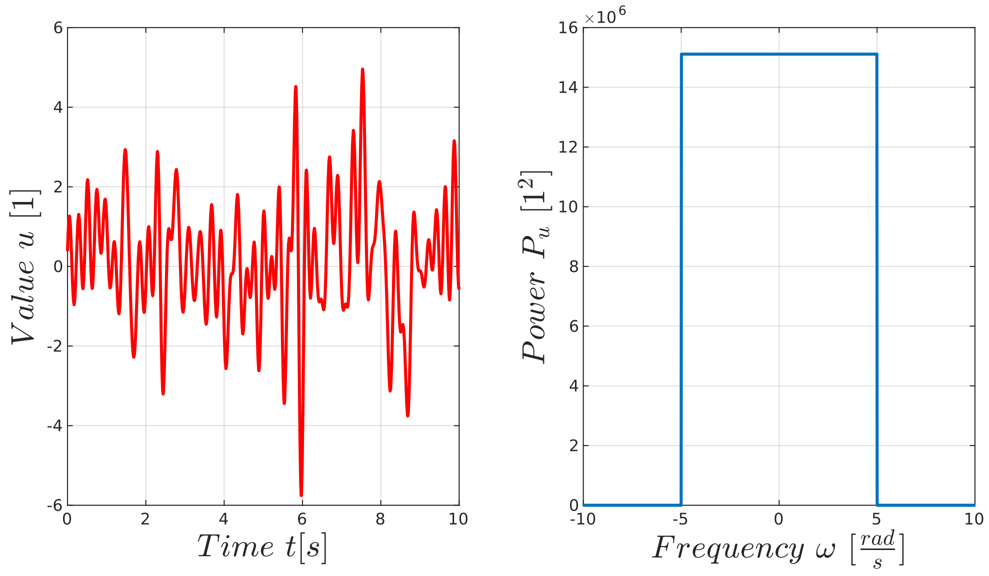


Fig. 16: Band-limited noise sent into input u_1 for identification purposes.

Three different band-limited noise signals $n_i,\ i = (0, 1, 2)$ are generated, one for each input. The input vector $\mathbf{u}$ is thus defined as

$$\mathbf{u}(t) = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} n_1(t) \\ n_2(t) \\ n_3(t) \end{bmatrix}. \tag{53}$$

7.2 Code implementation

In this subsection, I'll describe the part of the simulation implementation code that wasn't trivial, isn't in the documentation (yet) and that's specific to simulating dynamical systems in MATLAB. Before the simulation can start, we must set up the ODE function for the solver. MATLAB's ODE solver expects on its input an ODE function with an input in the form `(t, x)`. The noise process used for inputs must be defined as a *continuous* function of time `t` and embedded into the ODE function used for simulation. The band-limited noise was generated as a discrete time signal, so it must be transformed into a continuous function. This can be done for example using simple linear interpolation. The input can also be a function of the state `x`, if we wished to simulate a system with feedback control - this will be useful later. While these last few steps might look difficult, they're easy to implement using MATLAB's anonymous functions:

```
control_law = @(x) [0, 0, 0];
random_process = @(t) interp1(time, noise_bandlimited, t);
u_fun = @(t, x) (control_law(x) + random_process(t));
```

The `time` variable is an $N \times 1$ vector of timestamps and `noise_bandlimited` is an $N \times 3$ matrix, where each column represents the value of a noise process in time. The `control_law` anonymous function will be useful for implementing feedback, but for now it's kept "disconnected". The `interp1` function is a $1D$ row-wise interpolation function;

since `noise_bandlimited` has 3 columns, it will return 3 values. By adding up both anonymous functions into `u_fun`, we get a function that determines the values of all 3 u_i input variables based on the time `t` and state `x`.
The ODE function for the Lorenz system:

```
function dxdt = lorenz(t, x, u_fun, params)

u = u_fun(t, x);

dxdt = [params.sigma*(x(2)-x(1)) + 40*sign(u(1)),...
    x(1)*(params.rho - x(3)) - x(2) + 10*sin(u(1))*u(2),...
    x(1)*x(2) - params.beta*x(3) + u(3)*x(1)]';
```

Note that this `lorenz` function still isn't a function of only the time `t` and `x` as the solver requires. It has two additional inputs, the function `u_fun(t, x)` and a struct with system parameters. Both these additional inputs are constant once defined. To embed the inputs into the ODE function, define another anonymous function as:

```
params.sigma = 10;
params.beta = 8/3;
params.rho = 28;
odefun = @(t, x)lorenz(t, x, u_fun, params);
```

This `odefun` function is now in the correct form for the `ode45` solver.

7.3 Simulation

The simulation is now simple:

```
x0 = [1, 2, 3]'; % initial conditions
tspan = [0, t_end]; % time span
sol = ode45(odefun, tspan, x0); % Solve the system of ODEs

t = (0:dt:t_end)'; % define a time vector with constant time steps
x = deval(sol, t)'; % use the sol object to get the states at the particular times
```

The outputs of the simulation are the data with time and state trajectory measurements $\mathbf{X}$. The total number of time samples `N` is 2^{16}, and the sampling period `dt` is $0.001\,\mathrm{s}$, putting the total simulation time at $\mathrm{t}_{\mathrm{end}} = 2^{16} * \Delta t \approx 65\,\mathrm{s}$.
The data is then sent back into the `odefun` function and the `u_fun` function to get the state derivative measurements $\dot{\mathbf{X}}$ and the input variable measurements $\mathbf{U}$ respectively. All the data is then saved in `.csv` format for the identification phase, which is implemented in another language, Python.

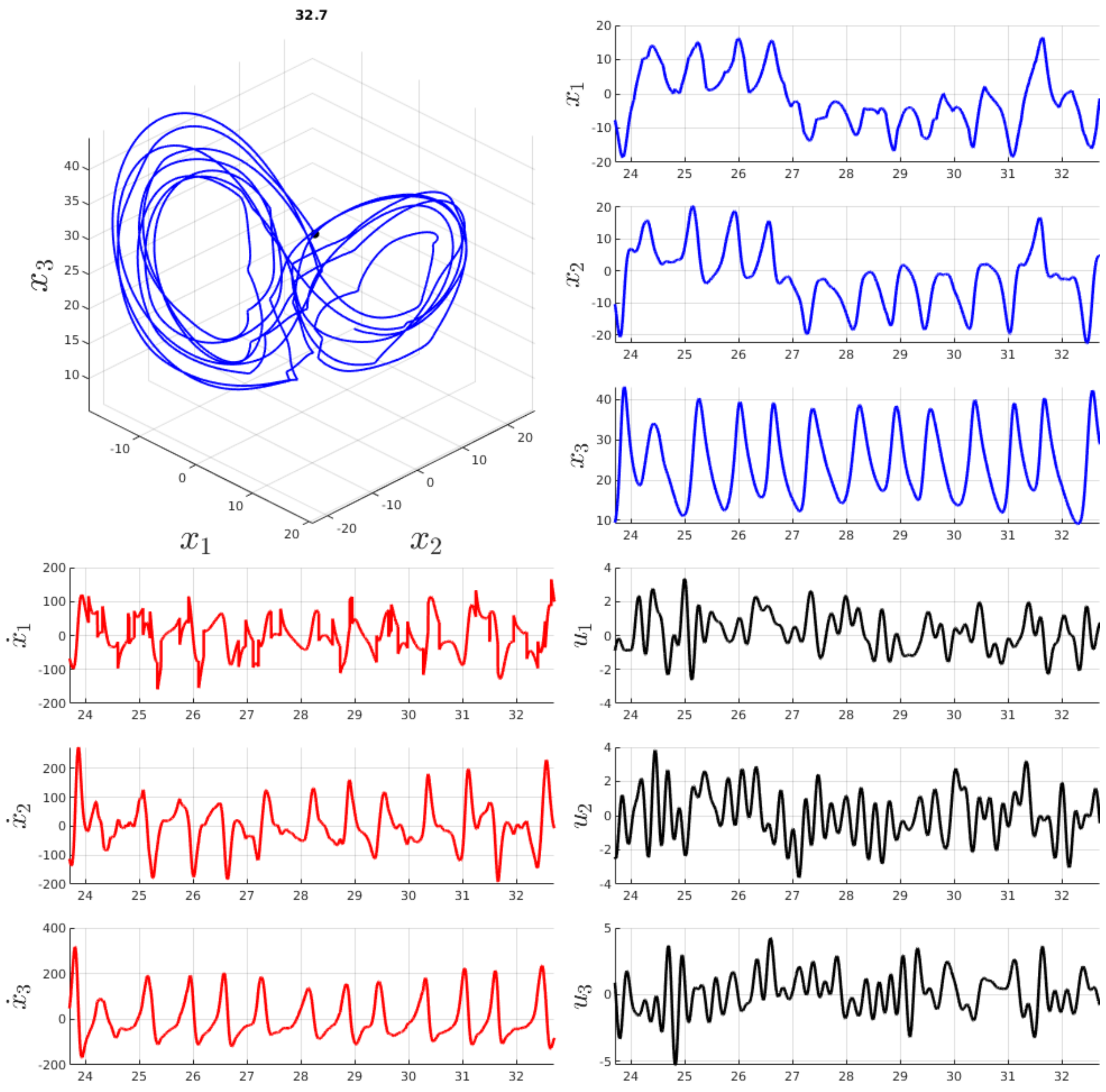


Fig. 17: Results of a simulation of a Lorenz system with external random process inputs. The full simulation is visualized as an animation at this link: https://git.io/JBctw.

7.4 Creating the function library

The library of candidate functions $\mathbf{\Theta}$ is then computed from the $\mathbf{x}$ and $\mathbf{U}$ simulation data. The first set candidate functions was chosen as all possible products of the state and input variables up to the 2nd order. The list of candidate function identifiers (column names) for this set:

```
Theta1.columns = [
    'u_1', 'u_2', 'u_3', 'x_1', 'x_2', 'x_3', 'u_1*u_1', 'u_1*u_2',
    'u_1*u_3', 'u_1*x_1', 'u_1*x_2', 'u_1*x_3', 'u_2*u_2', 'u_2*u_3',
    'u_2*x_1', 'u_2*x_2', 'u_2*x_3', 'u_3*u_3', 'u_3*x_1', 'u_3*x_2',
    'u_3*x_3', 'x_1*x_1', 'x_1*x_2', 'x_1*x_3', 'x_2*x_2', 'x_2*x_3',
    'x_3*x_3']
```

The second set of candidate functions was chosen as the Cartesian product of the set of all sine and cosine functions of inputs u_i and the set of all inputs u_1 and state variables x. The column names of this data matrix are:

```
Theta2.columns = [
    'sin(u_1)*u_1', 'sin(u_1)*u_2', 'sin(u_1)*u_3', 'sin(u_1)*x_1',
    'sin(u_1)*x_2', 'sin(u_1)*x_3', 'sin(u_2)*u_1', 'sin(u_2)*u_2',
    'sin(u_2)*u_3', 'sin(u_2)*x_1', 'sin(u_2)*x_2', 'sin(u_2)*x_3',
    'sin(u_3)*u_1', 'sin(u_3)*u_2', 'sin(u_3)*u_3', 'sin(u_3)*x_1',
    'sin(u_3)*x_2', 'sin(u_3)*x_3', 'cos(u_1)*u_1', 'cos(u_1)*u_2',
    'cos(u_1)*u_3', 'cos(u_1)*x_1', 'cos(u_1)*x_2', 'cos(u_1)*x_3',
    'cos(u_2)*u_1', 'cos(u_2)*u_2', 'cos(u_2)*u_3', 'cos(u_2)*x_1',
    'cos(u_2)*x_2', 'cos(u_2)*x_3', 'cos(u_3)*u_1', 'cos(u_3)*u_2',
    'cos(u_3)*u_3', 'cos(u_3)*x_1', 'cos(u_3)*x_2', 'cos(u_3)*x_3']
```

These two function libraries `Theta1` and `Theta2` were concatenated into one large candidate function library `Theta`. One extra function was generated for the candidate function $\mathrm{sgn}(u_1)$ and also added into `Theta`. This would've been enough, but since I only implemented the implicit form of the method, the identification object expects the state derivative measurements $\mathbf{X}$ to be present in the library, so they were also put in.

The previous sequence of operations gave us just the symbolic representations of our candidate functions. We have to recreate the numeric, discrete representation for the purpose of regression. The trigonometric terms, for example the term $\sin(u_1)$, will be recreated by taking the time-series of measurements of u_1 and applying the $\sin(u)$ function to every element of the vector.

$$u_2 = \begin{bmatrix} u_2[0] \\ u_2[1] \\ \vdots \\ u_2[N-2] \\ u_2[N-1] \end{bmatrix} \qquad \sin(u_2) = \begin{bmatrix} \sin(u_2[0]) \\ \sin(u_2[1]) \\ \vdots \\ \sin(u_2[N-2]) \\ \sin(u_2[N-1]) \end{bmatrix} \tag{54}$$

The candidate functions that are a product of other functions are created by taking the Hadamard (element-wise) product of their respective time series. For example the time-series vector representing the candidate function $\cos(u_2) * x_3$ will be calculated as:

$$\begin{aligned}\cos(u_2) * x_3 &= \begin{bmatrix} \cos(u_2[0]) \\ \cos(u_2[1]) \\ \vdots \\ \cos(u_2[N-2]) \\ \cos(u_2[N-1]) \end{bmatrix} \circ \begin{bmatrix} x_3[0] \\ x_3[1] \\ \vdots \\ x_3[N-2] \\ x_3[N-1] \end{bmatrix} = \\ &= \begin{bmatrix} (\cos(u_2[0])x_3[0]) \\ (\cos(u_2[1])x_3[1]) \\ \vdots \\ (\cos(u_2[N-2])x_3[N-2]) \\ (\cos(u_2[N-1])x_3[N-1]) \end{bmatrix} = \begin{bmatrix} (\cos(u_2)x_3)[0] \\ (\cos(u_2)x_3)[1] \\ \vdots \\ (\cos(u_2)x_3)[N-2] \\ (\cos(u_2)x_3)[N-1] \end{bmatrix}\end{aligned} \tag{55}$$

Where $\circ$ is the Hadamard product operator.
The result is a matrix representing the function library $\boldsymbol{\Theta} \in \mathbb{R}^{N \times m}$, where each column is a candidate function θ_i, each row is a measurement at discrete time k, N is the number of time-samples and m is the number of candidate functions.

$$\begin{aligned}\boldsymbol{\Theta} &= \begin{bmatrix} \theta_0 & \theta_1 & \dots & \theta_{m-2} & \theta_{m-1} \end{bmatrix} = \\ &= \begin{bmatrix} \theta_0[0] & \theta_1[0] & \dots & \theta_{m-2}[0] & \theta_{m-1}[0] \\ \theta_0[1] & \theta_1[1] & \dots & \theta_{m-2}[1] & \theta_{m-1}[1] \\ \vdots & \vdots & \dots & \vdots & \vdots \\ \theta_0[N-2] & \theta_1[N-2] & \dots & \theta_{m-2}[N-2] & \theta_{m-1}[N-2] \\ \theta_0[N-1] & \theta_1[N-1] & \dots & \theta_{m-2}[N-1] & \theta_{m-1}[N-1] \end{bmatrix}\end{aligned} \tag{56}$$

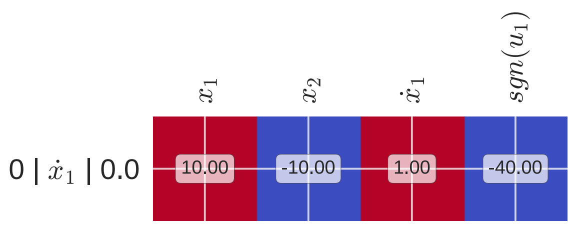
7.5 Identification using clean data

The candidate function matrix $\boldsymbol{\Theta}$ contains all the information needed to start the identification phase itself. The Lorenz system has three state variables x_1, x_2, x_3. To find the model of the system, we need to find their derivatives $\dot{x_1}, \dot{x_2}, \dot{x_3}$. These derivatives are also contained in the $\boldsymbol{\Theta}$ library, because my implementation of the identification method uses the implicit formulation of the identification process.

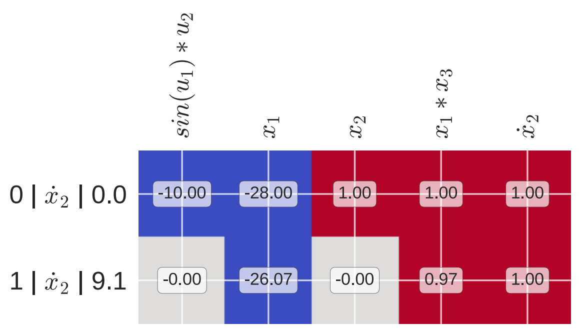
A separate identification procedure is done for each state derivative. The procedure starts by removing the candidate functions θ that aren't relevant for the state derivative that's being identified. In this case, it'll be only the other state derivatives. When identifying $\dot{x_1}$, candidate functions $\dot{x_2}$ and $\dot{x_3}$ are dropped from the library and the library is sent to the object that does the identification. The object finds the parameters $\boldsymbol{\xi}$ of implicit models according to the equation (46). Because we know that the dynamics can be expressed as a linear combination of candidate functions, we don't need to solve the implicit form of the problem. Instead of making the identification object make a guess for every column θ_i, we can define the guess to be only the relevant state derivative column. This way, the identification procedure is identical to the original explicit SINDy formulation (8), where the state derivative is the target variable directly.

The identification object uses sequentially energy-thresholded least squares algorithm, which I introduced in 3.2.5, to find the solution $\boldsymbol{\xi}$. The regression algorithm has one hyperparameter, λ_R, which controls the sparsity of the solution. The identification object requires us to define a set of hyperparameter values to use for regression. A separate model is created for each specified hyperparameter. I defined 10 hyperparameter values, ranging from 0.001 (least sparse) to 0.01 (most sparse). Therefore, 10 models will be generated for each state variable.

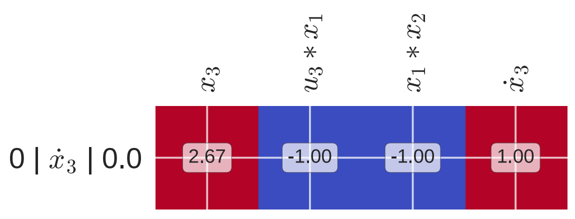
The unique models for each state variable are shown in the Figures 18. Each row represents a single implicit model. The row labels contain the model's index, the candidate function used as the guess function, and the training error metric (RMSE, root mean square error). Each column represents a candidate function θ_i and its respective parameter $\boldsymbol{\xi}_i$. Candidate functions that weren't active in any of the identified models aren't visualized.



(a) Identified implicit models for $\dot{x}_1$



(b) Identified implicit models for $\dot{x}_2$



(c) Identified implicit models for $\dot{x}_3$

Fig. 18: Identification results for the Lorenz system simulated with feed-forward input signals. The training data contained no noise, and derivatives were computed exactly from the analytical model.

7.6 Identification using noisy data

Real measurements contain noise. I'll repeat the identification process, but I'll add white noise with standard deviation of 0.2 to each measurement vector x_i. Measurements of the derivatives $\dot{x}_i$ will also be estimated from measurements of x_i using numerical differentiation. First, it's necessary to get rid of the high frequency part of the noise present in the measurements. Otherwise, numerical differentiation would result in an extremely noisy signal. The filter settings for each state variable are shown in Figure 19.

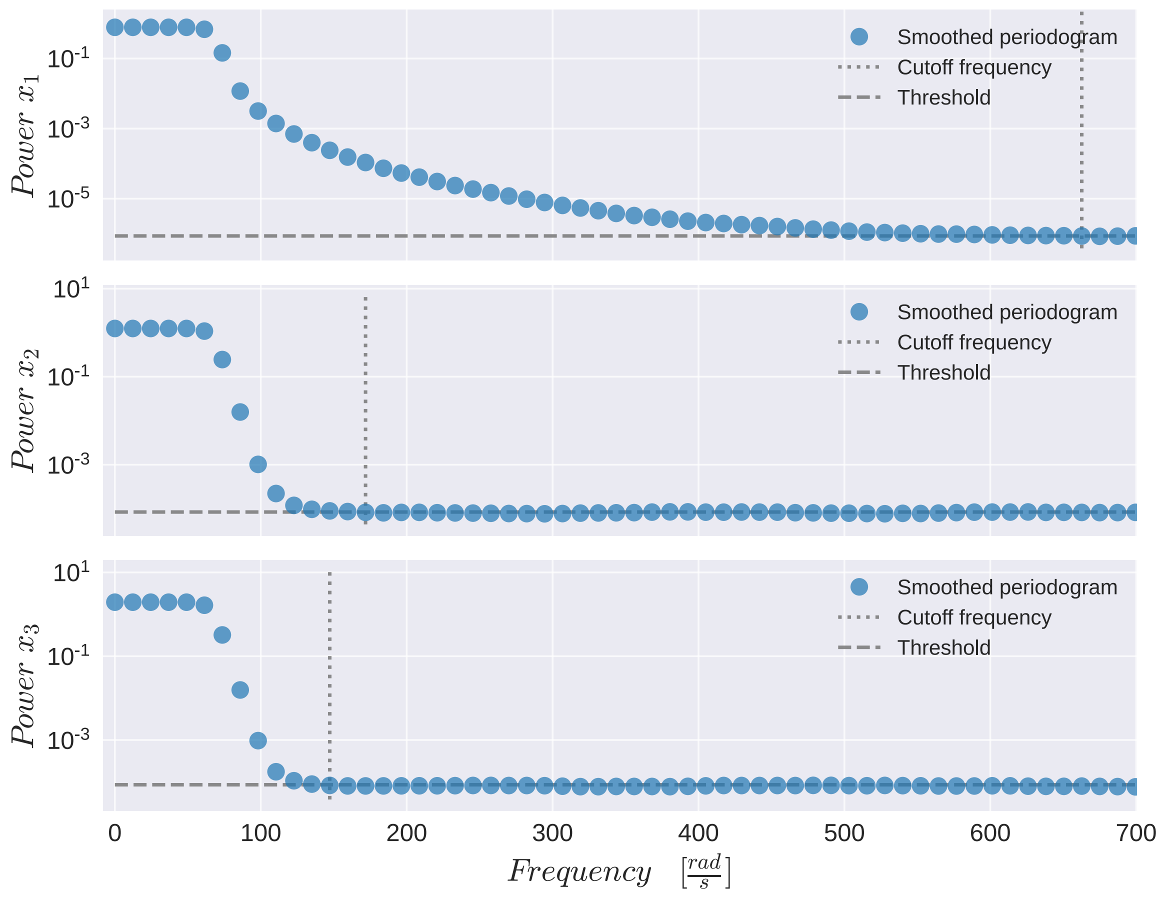


Fig. 19: Spectral filter settings for the noisy measurements of x_i.

The x_1 signal and its respective model $\dot{x}_1$ will be the most difficult to identify due to the discontinuity caused by the $\mathbf{sgn}(u_1)$ term. This term causes first order discontinuities (jumps) in the measurements of the derivative $\dot{x}_i$. The discontinuities in time-domain lead into infinite bandwidth in frequency domain. This is an issue, because the high frequency information signal is relatively weak compared to the noise. High frequencies need to be filtered out for numerical differentiation, otherwise we'd be mostly differentiating noise. This leads into the estimated measurements of $\dot{x}_1$ being quite off from the real values, as shown in Figure 21.

Next, the function library $\mathbf{\Theta}$ is constructed from the measurements and the derivative estimates and used as a training dataset for the identification. The identified models are shown in Figure 22.

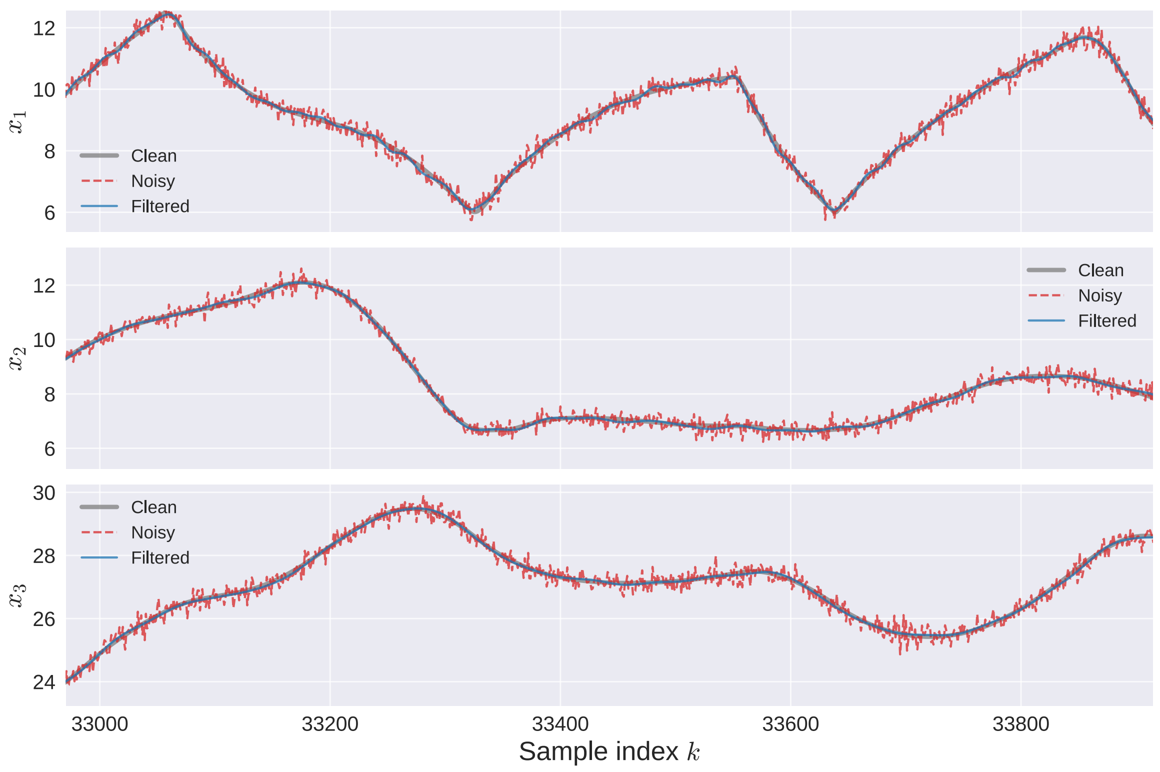


Fig. 20: Comparison of clean, noisy and filtered measurements of x_i.

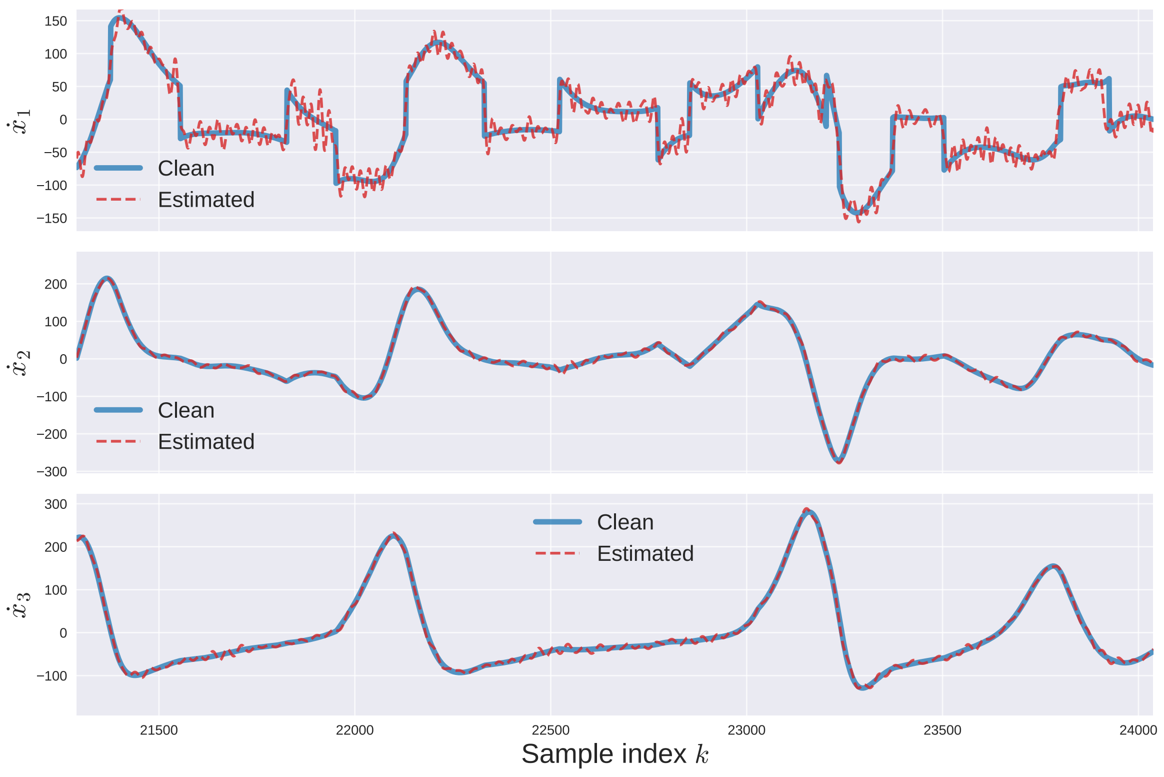
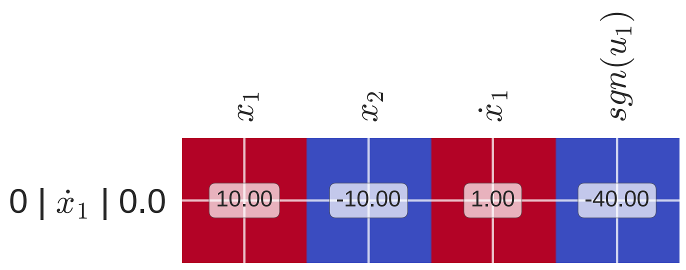
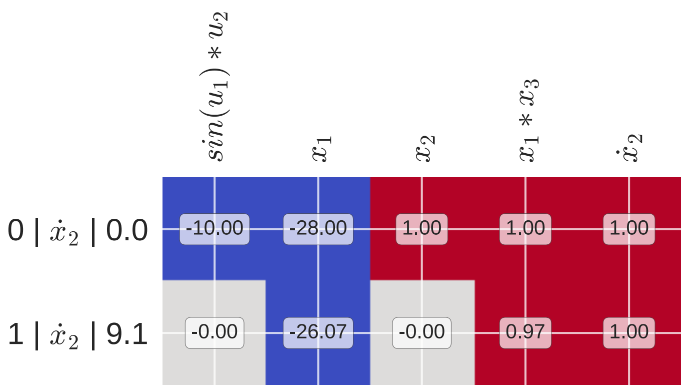


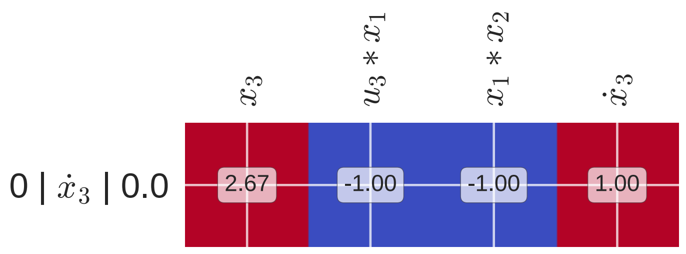
Fig. 21: Comparison of clean and estimated values of $\dot{x}_i$. The estimated values were computed from filtered measurements x_i using spectral differentiation.



(a) Identified implicit models for $\dot{x}_1$



(b) Identified implicit models for $\dot{x}_2$



(c) Identified implicit models for $\dot{x}_3$

Fig. 22: Identification results for the Lorenz system simulated with feed-forward input signals. The measurements of state variables contained significant noise, which lead into poor state derivative estimate accuracy.

7.7 Validation

The simulated dataset was first split into two data sets, one used for training, and the other for validation. The validation data set was kept clean, with no noise and exact derivatives. The model derived for earlier is first validated using the validation set. The derivatives $\dot{\mathbf{x}}$ are first estimated using the state $\mathbf{x}$ and $\mathbf{u}$. The results are in Figure 23.

The model's state derivative estimations are quite accurate, being very close to the real values. The reference model (used to generate the data) and the model identified from noise will further be validated by simulation. Both models will be simulated, from an identical initial state and both receiving the same input signal. The trajectories are visualized in Figure 24. The trajectories start close together, but later decouple.

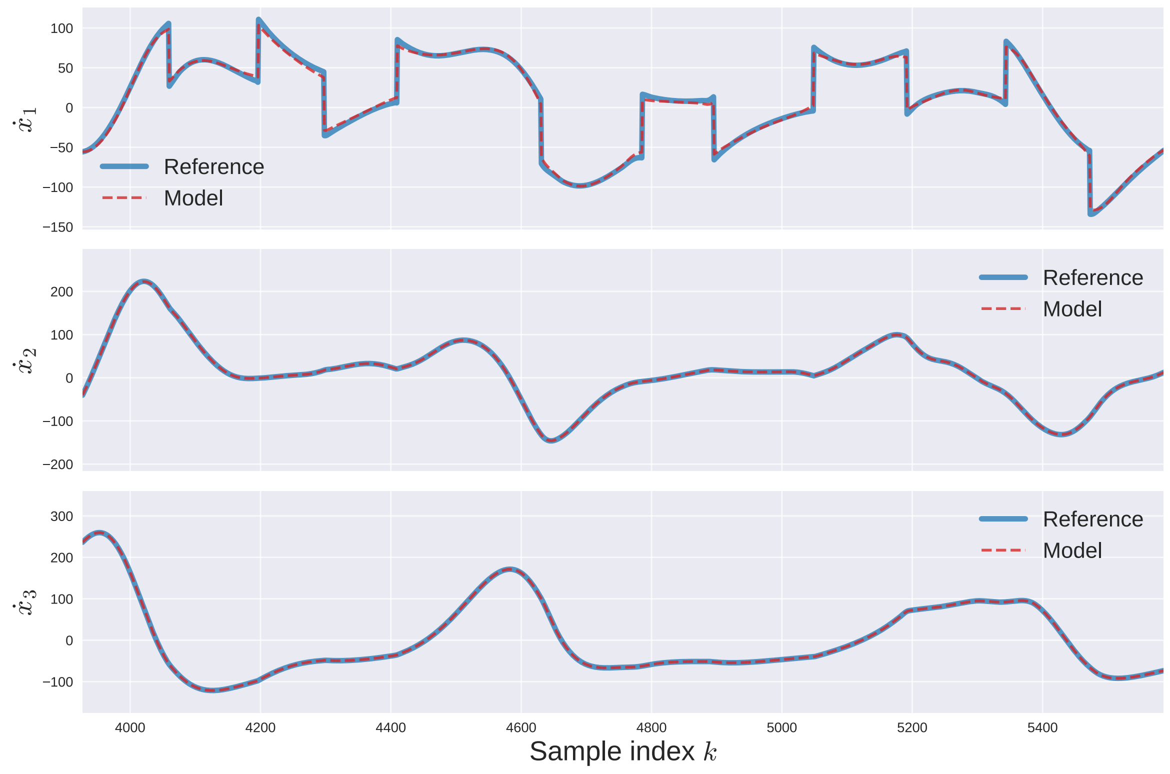


Fig. 23: Comparison of clean $\dot{x}_i$ from the validation set and the $\dot{x}_i$ estimated using the model identified from data.

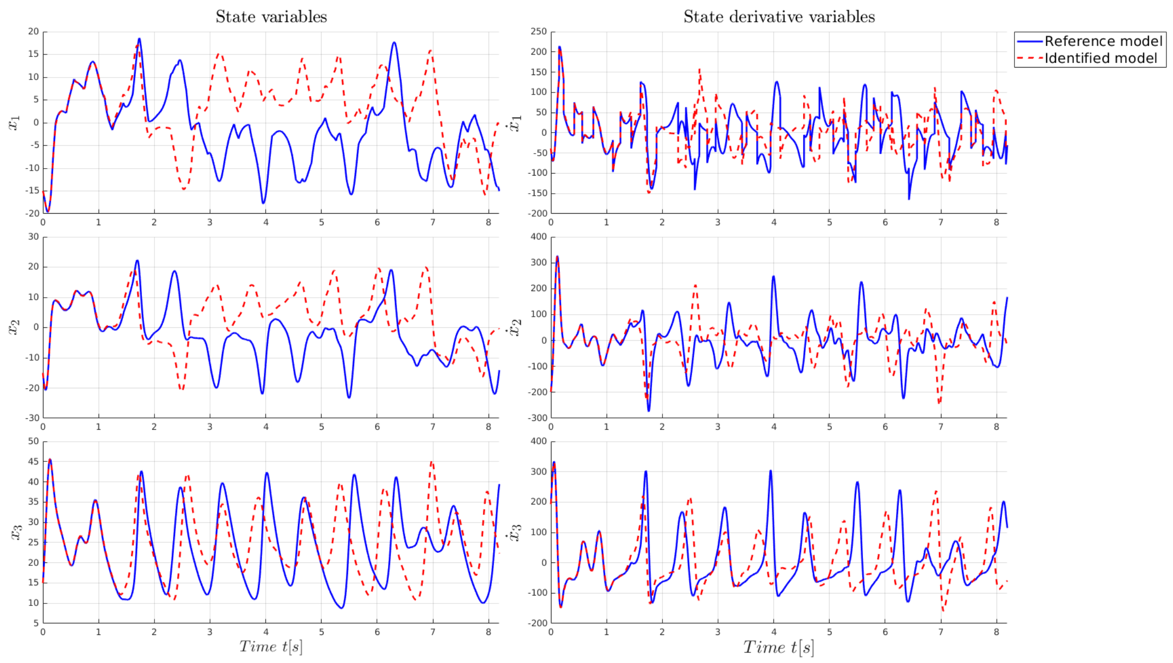


Fig. 24: Comparison of the trajectories generated by the reference and the identified model. Both models start at the same initial condition and receive the same input $\mathbf{u}$. The input signals are random band-limited processes, but they're not visualized. Full animation: https://git.io/JBSU3

7.8 Discussion

The exact model of the simulated system was fully retrieved from clean simulation data. The training error for all three system equations was 0. Training error doesn't test generalization capability, and models should always be evaluated on a test set that was withheld from training. The sparsity condition promotes simplicity, which in turn promotes generalization capability. If sparse models were found and had a good training error, it's very likely they're able to generalize well.

The model identified from clean test data wasn't validated, since the entire identification process was idealized and served only to confirm that the method works. The simulation data was completely clean. It didn't contain any noise and the state derivatives were computed from the analytical system equations directly.

Noise was then added to the state measurements and the state derivative measurements used for training were estimated from the measurements to create the training data set. The most accurate models were accurate for all state derivative models except for $\dot{x}_1$, where the correct model was the *second* most accurate when evaluated on the training data. The model for $\dot{x}_1$ was the most difficult to identify because of the first order discontinuities in the $\dot{x}_1$ measurements caused by the $\mathrm{sgn}(u_1)$ function.

8 Model identification of the Lorenz system with feedback external inputs

8.1 Definition

In this section, the system's natural dynamics are the same as in the previous section, but the external inputs will enter the state derivative equations directly.

$$\dot{x}_1 = 10x_2 - 10x_1 + u_1 \tag{57a}$$

$$\dot{x}_2 = 28x_1 - x_1x_3 - x_2 + u_2 \tag{57b}$$

$$\dot{x}_3 = x_1x_2 - \frac{8}{3}x_3 + u_3. \tag{57c}$$

The purpose of this simplification is to demonstrate the identification of a feedback-controlled system. If u_1 entered the dynamics as an argument of a non-linear function, for example as $\mathrm{sqn}(u_1)$, and u_1 was defined by the feedback control law $u_1 = x_1 - 15$, then the control term in the dynamics would be $\mathrm{sqn}(x_1 - 15)$. The argument of the sqn function would be shifted by 15. This would mean that the function $\mathrm{sqn}(x_1 - 15)$ would have to be present in the candidate function library $\mathbf{\Theta}$. Identifying terms with shifted arguments is not feasible. To identify the argument-shifted term, the function library have to contain the function with the exact shift. Without exact knowledge of the argument shift, it'd have to be guessed by including the function with many different argument shifts and seeing which one is correct. The library would grow too large and the identification would become poorly conditioned.

The identification process in this case will be made more difficult by three things:

1. I won't use the exact state derivatives computed from the system equations. Instead they'll have to be calculated using numerical differentiation from the state measurements $\mathbf{X}$.

2. I'll add noise to the state measurements $\mathbf{X}$. This will make directly computing state derivatives $\dot{\mathbf{X}}$ from $\mathbf{X}$ impossible, due to the high-pass nature of numerical differentiation. The state measurements $\mathbf{X}$ will have to be filtered.

3. The input values $\mathbf{u}$ during the simulation won't be defined as pure random processes, they will contain state feedback. State feedback effectively changes the system dynamics. If the inputs were defined as pure feedback, it would be impossible to tell natural dynamics apart from effects of external forcing. This problem will be discussed further in the next subsection.

8.2 Feedback forcing loop

In the previous section, the input was defined as a random noise process to identify the effects of external inputs on the system. In some real world applications, we might not have the option to send arbitrary signals as inputs. Sometimes, the inputs have to be also used to control (stabilize) the system continuously, even during the data collection phase.

In this example, the inputs will contain state feedback terms. State feedback effectively changes the dynamics of the system. If the inputs $\mathbf{u}$ weren't measured during the simulation or if the input measurements $\mathbf{U}$ weren't included in the candidate function library $\mathbf{\Theta}$, it would be completely impossible to tell apart the effects of natural dynamics from external forcing. The regression problem would become poorly conditioned. If the control law defined the first input as $u_1 = x_1 - 15$, then the measurement of u_1 would simply consist of y-axis shifted measurements of x_1. This would make both candidate functions x_1 and u_1 perfectly correlated, the condition number κ would explode to ∞. For this reason, the inputs $\mathbf{u}$ cannot be defined purely by the feedback control law for data collection. Let's define a general input u_i as the sum of the control law u_{ic} and some random process u_{in} as

$$u_i(t, \mathbf{x}) = u_{ic}(\mathbf{x}) + u_{in}(t). \tag{58}$$

The time dependent u_{in} random noise process helps to de-correlate the input measurements from the state measurements, allowing combined identification of both the natural dynamics and the effects of external forcing. The relative power of both u_{ic} and u_{in} signals also plays an important role. If the random process $u_{in}(t)$ is strong compared to the control law $u_{ic}(\mathbf{x})$, the measurement vector of u_i will be strongly de-correlated from the state measurements. But this benefit comes at a cost; as the control law is relatively weak, the system will be poorly controlled during data collection, which could have consequences significantly worse than "bad data".

The inputs will be defined during the simulation as

$$\mathbf{u} = \begin{bmatrix} 0.125(x_1 - 20) + n_1(t)) \\ 0.125(x_2 - 20) + n_2(t)) \\ 0.125(x_3 - 24) + n_3(t)) \end{bmatrix} \tag{59}$$

where $n_i(t)$ are band-limited noise processes, as in the previous section. The first few seconds of the input signals during simulation is shown in the Figure 25.

8.3 Simulation and data pre-processing

The dynamical system is simulated with the input signals $\mathbf{u}$ defined by both the control law and the random noise process. The simulation procedure is similar as in the last section. There's clean simulation data on the simulation's output. To make the identification procedure more realistic, I'll add white noise with standard deviation

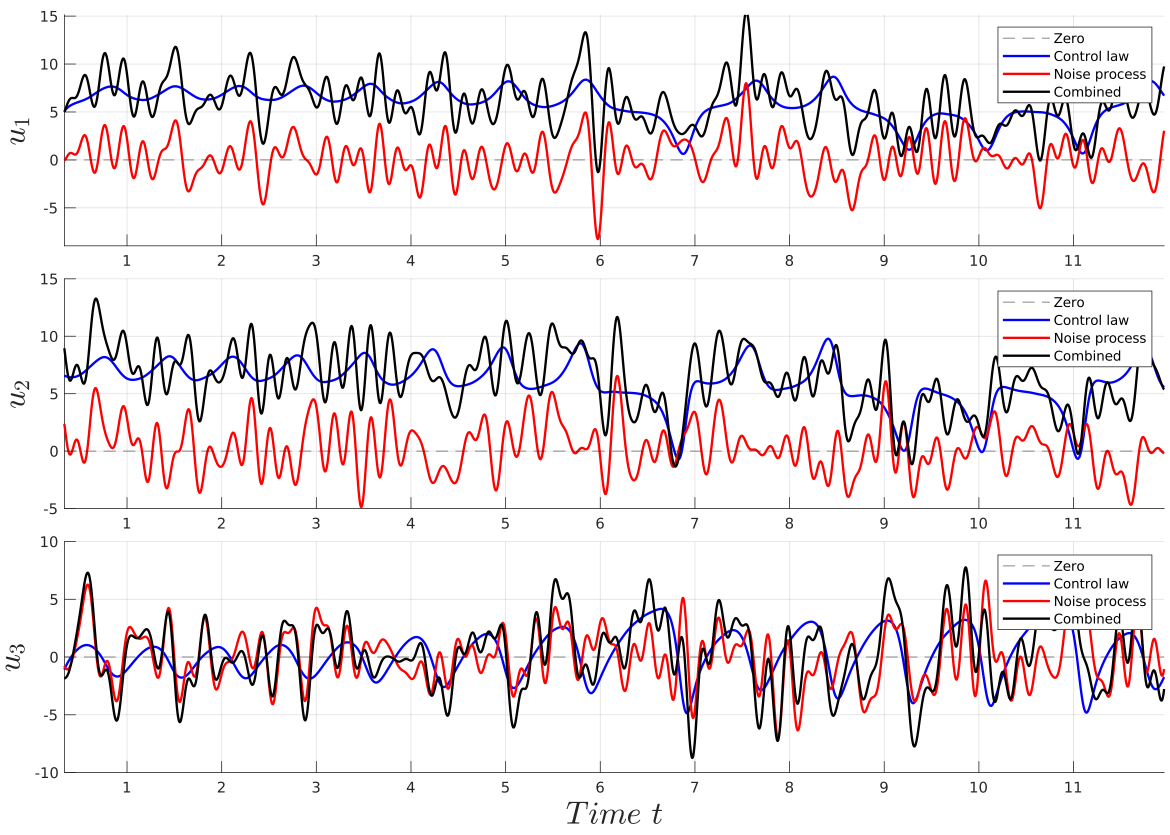
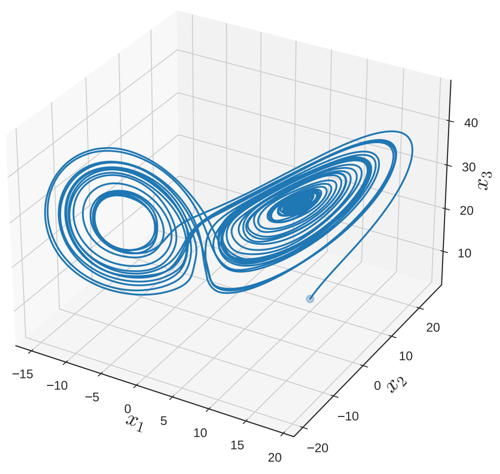
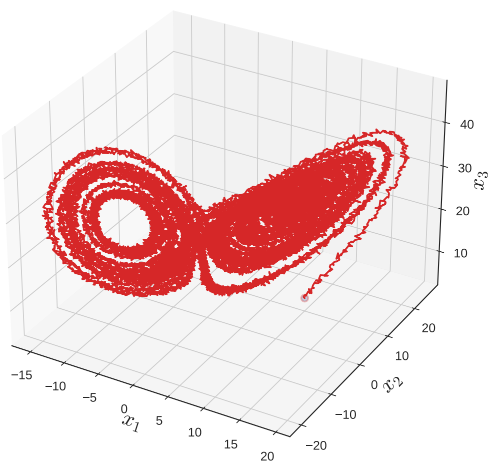


Fig. 25: The input signals as a sum of the control law and random noise process.

$\sigma = 0.25$ to all state measurements $\mathbf{X}$. The clean and noisy state trajectories $\mathbf{X}$ are shown in the next figure.



(a) The clean state trajectory $\mathbf{X}$.



(b) The noisy state trajectory $\mathbf{X}$.

Fig. 26: Comparison of generated state trajectory $\mathbf{X}$ with and without noise. A visualization of the full simulation: https://git.io/JBctq.

Another problem is getting the state derivative measurements $\dot{\mathbf{X}}$. In the previous section, I used the clean derivative measurements, which were generated using the real system equations. The point of system identification is discovering exactly these equations from data, so using them for pre-processing doesn't make sense chronologically. Instead, they'll have to be calculated from the state measurements $\mathbf{X}$ using numerical differentiation. Because numerical differentiation is sensitive to high-frequency noise, the state measurements $\mathbf{X}$ will first need to be low-pass filtered. I'll be doing this using the spectral filtering method described earlier in 4.2.4. The filter's cutoff frequencies and state measurements are shown in the next figures.

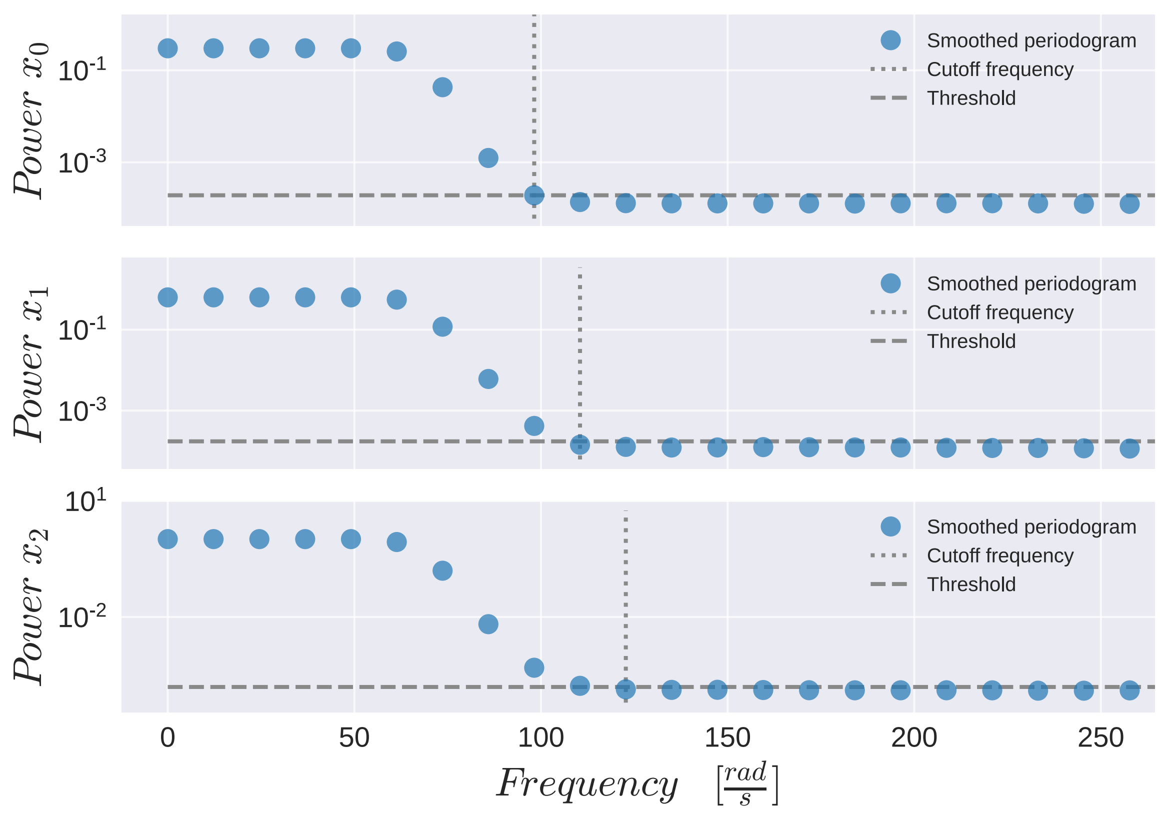
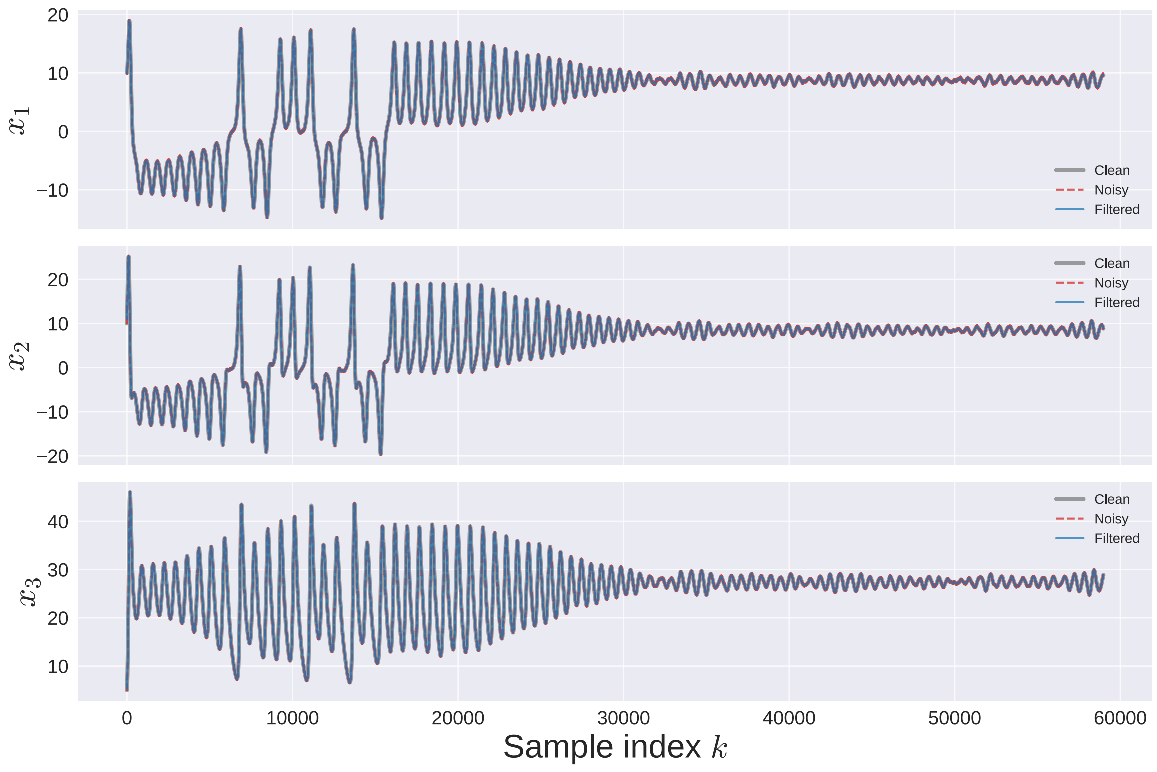
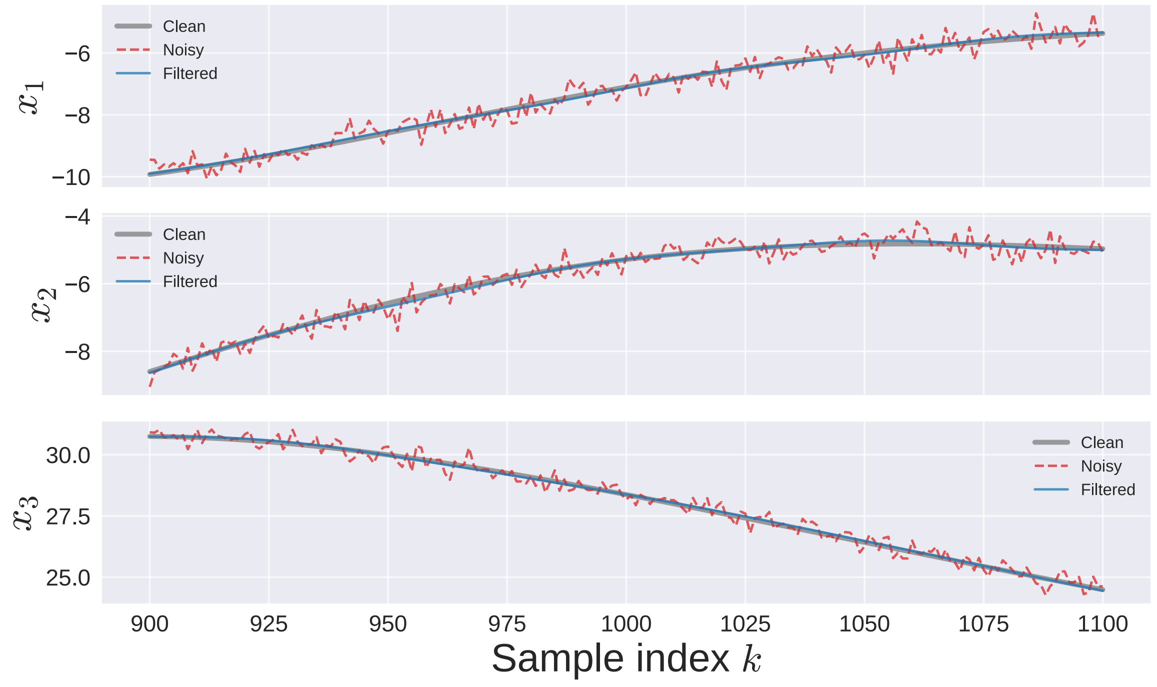


Fig. 27: Filter cutoff frequency setting from smoothed periodograms calculated from $\mathbf{X}$.



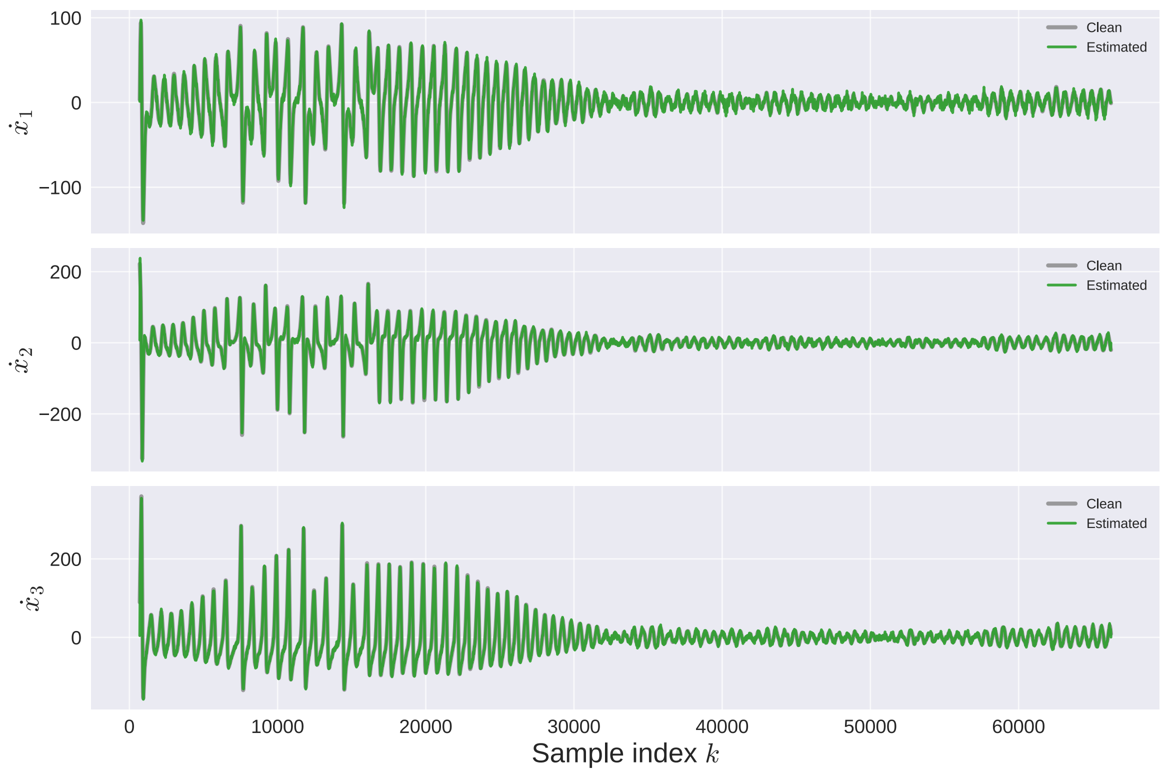
(a) The full state measurements.



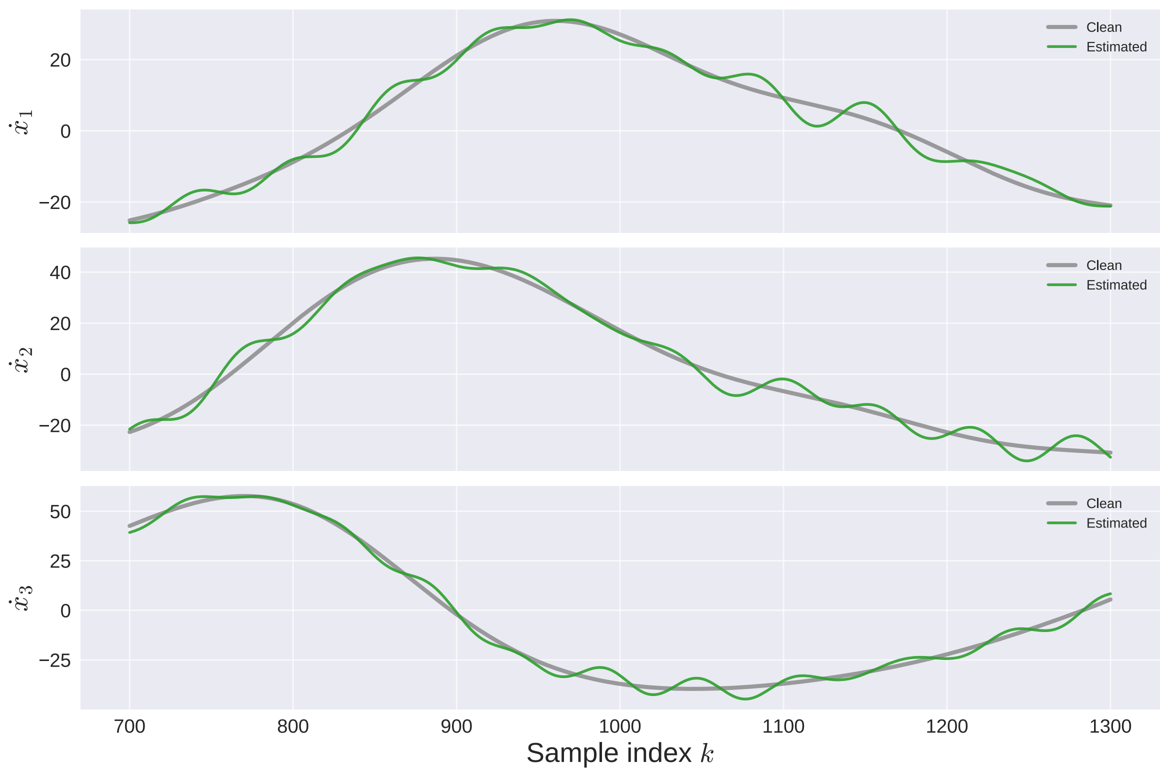
(b) The full state measurements, zoomed in at a portion of the signal.

Fig. 27: A comparison of clean, noisy and filtered state measurements $\mathbf{X}$.

Note that the filtered state measurements are nearly identical to the real, clean state measurements. The next step is calculating the state derivative measurements $\dot{\mathbf{X}}$ using spectral differentiation as discussed in 4.1.2. They will be calculated directly from the filtered $\mathbf{X}$ signals shown earlier.



(a) The full state state derivative measurements.



(b) The full state derivative measurements, zoomed in at a portion of the signal.

Fig. 27: A comparison of clean state derivative measurements $\dot{\mathbf{X}}$ and $\dot{\mathbf{X}}$ computed from filtered $\mathbf{X}$.

The estimated state derivatives oscillate quite a lot compared to the clean signals. The cutoff frequency was picked based on the lowest frequency at which the smoothed periodogram hits the “noise ceiling”. Spectral differentiation is effectively multiplication by $i\omega$ in the frequency domain, which corresponds to high-pass filter behaviour. This means that when estimating spectral derivatives, the highest frequency in the underlying signal will have a strong effect on the result. By picking the cutoff frequency as the frequency at which the noise ceiling is hit, the highest frequency in the filtered signal will have a (spectral) signal-to-noise ratio close above 1; the noise amplitude will be relatively close to the signal's amplitude. The high-pass nature of differentiation means that this signal frequency will have a strong weight on the results. For this reason, it's better to over-filter the signal before differentiation, so that the highest frequency still has a high signal-to-noise ratio. I offset the original cutoff frequency by 3 frequency bin ranges to the lower frequencies. Note that the periodogram was computed using Welch's method, so the frequency resolution is lower - a bin range is significantly higher than $\frac{f_{sampling}}{\#samples}$.

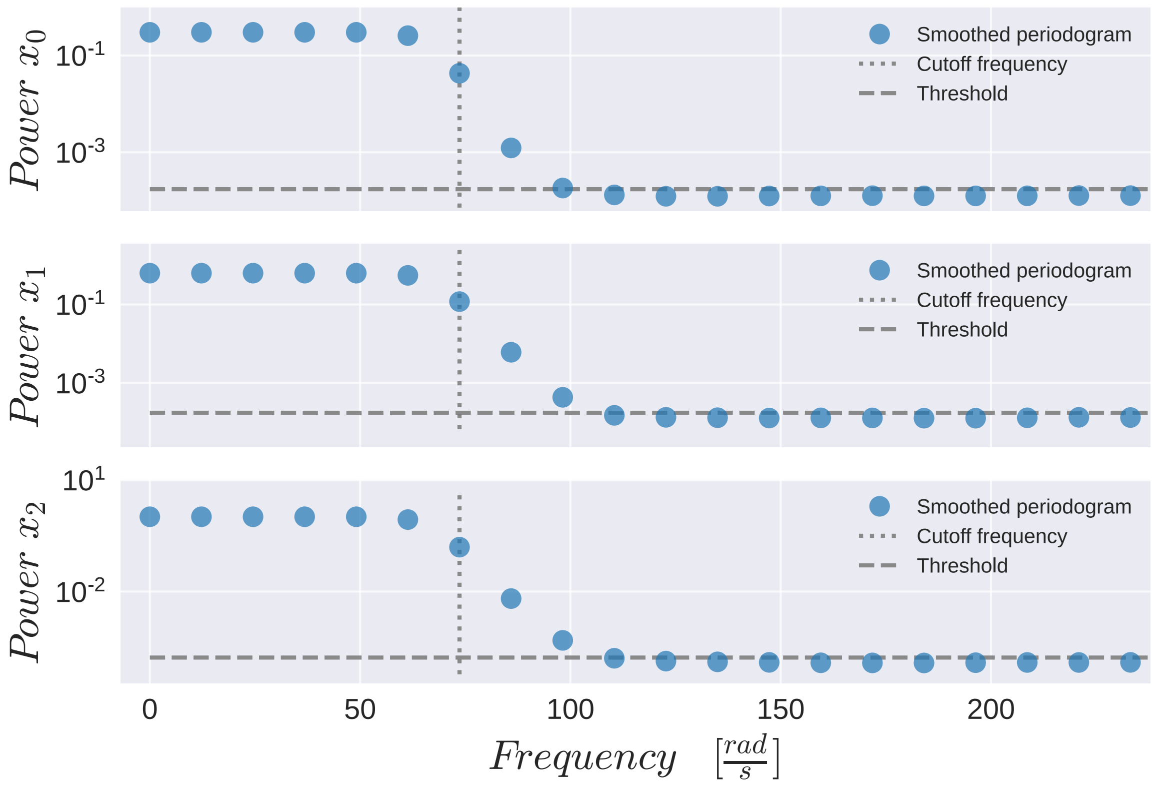


Fig. 28: Cutoff frequencies after being offset to lower frequencies by 3 frequency bins.

This filter is then used to again filter the noisy state measurements $\mathbf{X}$. The results are visually identical. The difference becomes significant when the signals are used to again compute $\dot{\mathbf{X}}$.

The data for regression doesn't have to be perfect. In fact, the identification was successful and yielded similar results for both cases. In any case, less noise is always better.

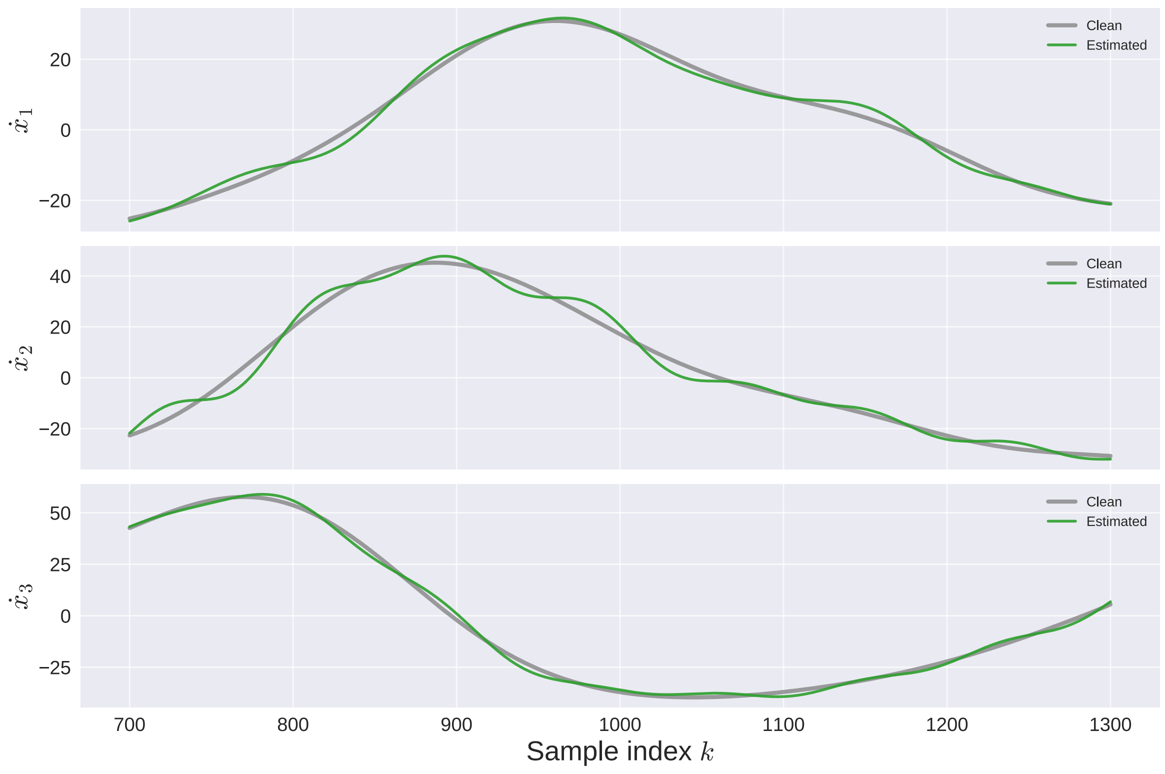


Fig. 29: State derivative measurement estimates $\dot{\mathbf{X}}$ computed from over-filtered $\mathbf{X}$.

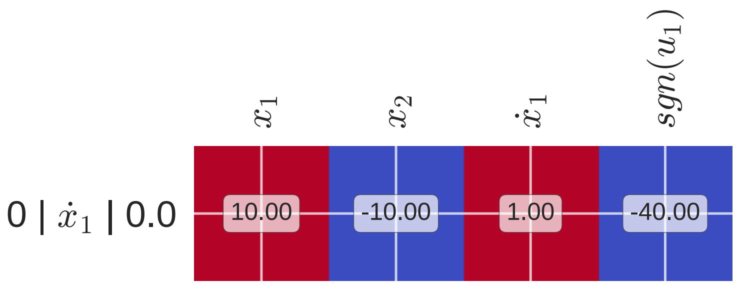
8.4 Identification

The candidate functions for the identification were defined as

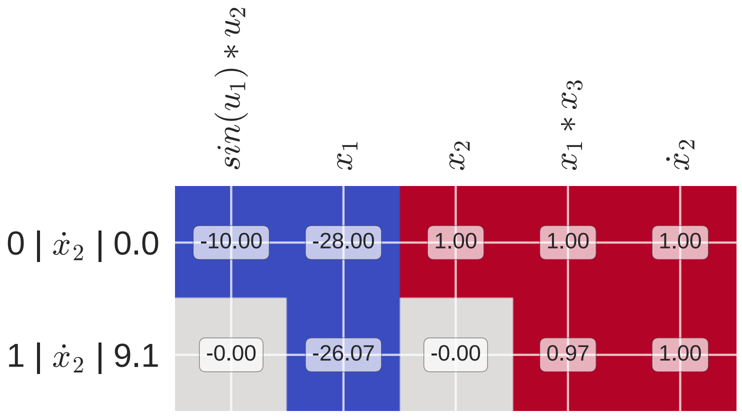
```
Theta.columns = [
'1', 'u_1', 'u_2', 'u_3', 'x_1', 'x_2', 'x_3', 'u_1*u_1', 'u_1*u_2',
  'u_1*u_3', 'u_1*x_1', 'u_1*x_2', 'u_1*x_3', 'u_2*u_2', 'u_2*u_3',
  'u_2*x_1', 'u_2*x_2', 'u_2*x_3', 'u_3*u_3', 'u_3*x_1', 'u_3*x_2',
  'u_3*x_3', 'x_1*x_1', 'x_1*x_2', 'x_1*x_3', 'x_2*x_2', 'x_2*x_3',
  'x_3*x_3', 'dx_1', 'dx_2', 'dx_3'].
```

The library is significantly smaller than the last time, where it included trigonometric and other functions. More candidate functions means more model parameters. When the regression algorithm has more parameters, it has more degrees of freedom, which often leads to overfitting; fitting the noise in the data. This means that in the presence of noise, the number of functions in the candidate function library $\boldsymbol{\Theta}$ cannot grow arbitrarily.
The discovered implicit models are shown in the next figure.

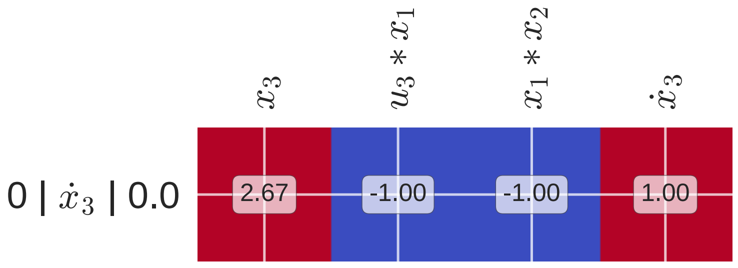
In all cases, the identified models contain exactly the correct terms. Due to noise present in the training data, the parameters are slightly off.



(a) Identified implicit models for $\dot{x}_1$



(b) Identified implicit models for $\dot{x}_2$



(c) Identified implicit models for $\dot{x}_3$

Fig. 28: Identification results for the Lorenz system simulated with feedback input signals. The state trajectories $\mathbf{X}$ contained significant noise, derivatives $\dot{\mathbf{X}}$ were computed using numerical differentiation from filtered $\mathbf{X}$.

Table 1: The system equation of the real model compared to the identified model.

$\dot{x}_i$	Reference model	Identified model
$\dot{x}_1$	$10x_2 - 10x_1 + u_1$	$9.99971x_2 - 10.00047x_1 + 0.99567u_1$
$\dot{x}_2$	$28x_1 - x_1x_3 - x_2 + u_2$	$27.71757x_1 - 0.99266x_1x_3 - 0.91848x_2 + 1.00354u_2$
$\dot{x}_3$	$x_1x_2 - \frac{8}{3}x_3 + u_3$	$0.99992x_1x_2 - 2.66735x_3 + 0.99746u_3$

8.5 Validation

During the identification phase, the models were evaluated on the training data. To test the models' generalization capability, they should be evaluated on data that the model hasn't seen during the regression.
First, let's test the model's ability to estimate the state derivative $\dot{\mathbf{x}}$ given a state $\mathbf{x}$ and input $\mathbf{u}$. This derivative prediction is to a continuous model as one-step prediction is to a discrete model. I ran the Lorenz simulation again, from a different initial state and using different random noise process on the input. Then I used the state measurements $\mathbf{X}_{val}$, input measurements $\mathbf{U}_{val}$ to calculate the state derivative estimates $\dot{\mathbf{X}}_{est}$ using the identified model. These estimates $\dot{\mathbf{X}}_{est}$ are then compared to the state derivative measurements $\dot{\mathbf{X}}_{val}$.

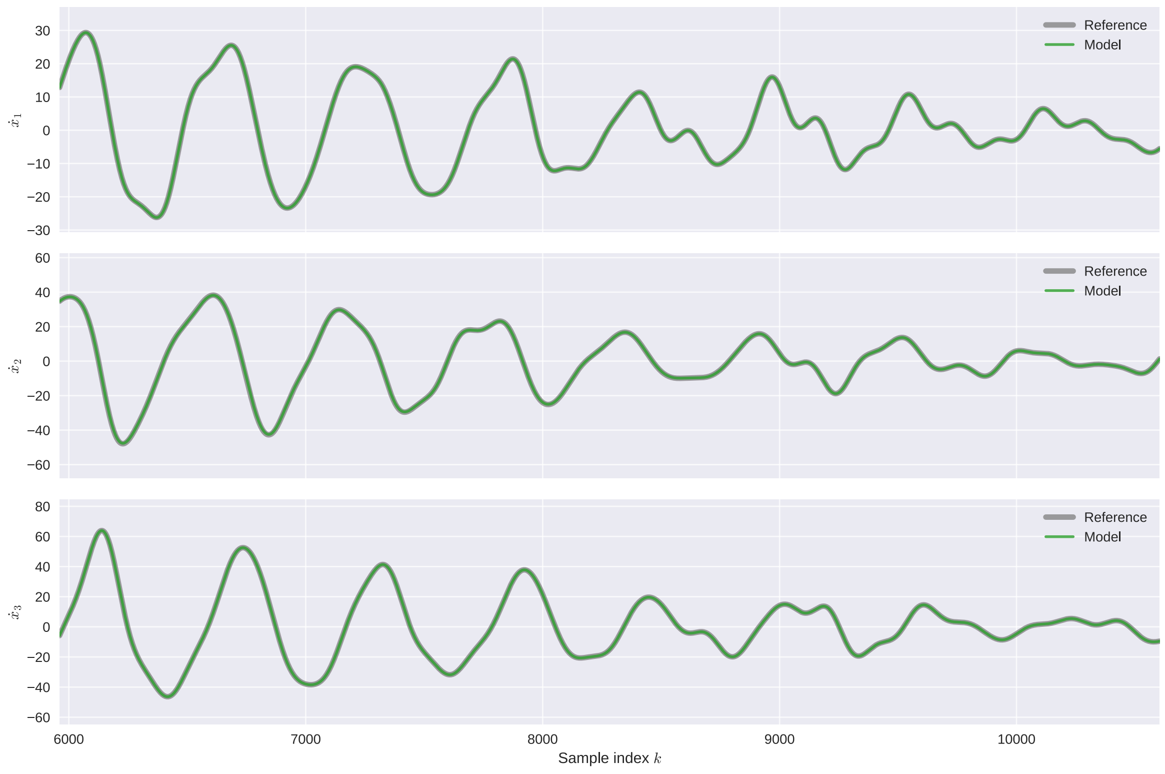


Fig. 29: Comparison of state derivatives $\dot{\mathbf{X}}_{est}$ estimated using the identified model and the reference $\dot{\mathbf{X}}_{val}$ validation data.

The estimates are nearly identical to the real values, since the model parameters are very accurate. For each state derivative $\dot{x}_i$, I calculated the error as the difference between real values and the values estimated by the model.

From the error signals above, I calculated the root-mean-square-error metric, RMSE:

Table 2: Validation RMSE for each identified state derivative $\dot{x}_i$.

$\text{RMSE}(\dot{x}_1 \vert \boldsymbol{\xi})$	2.33E-2
$\text{RMSE}(\dot{x}_2 \vert \boldsymbol{\xi})$	1.59E-1
$\text{RMSE}(\dot{x}_3 \vert \boldsymbol{\xi})$	2.34E-2

Curiously, the RMSE values are in each case significantly lower than the same metrics calculated from the training data as shown in the results visualization in Figure 28.

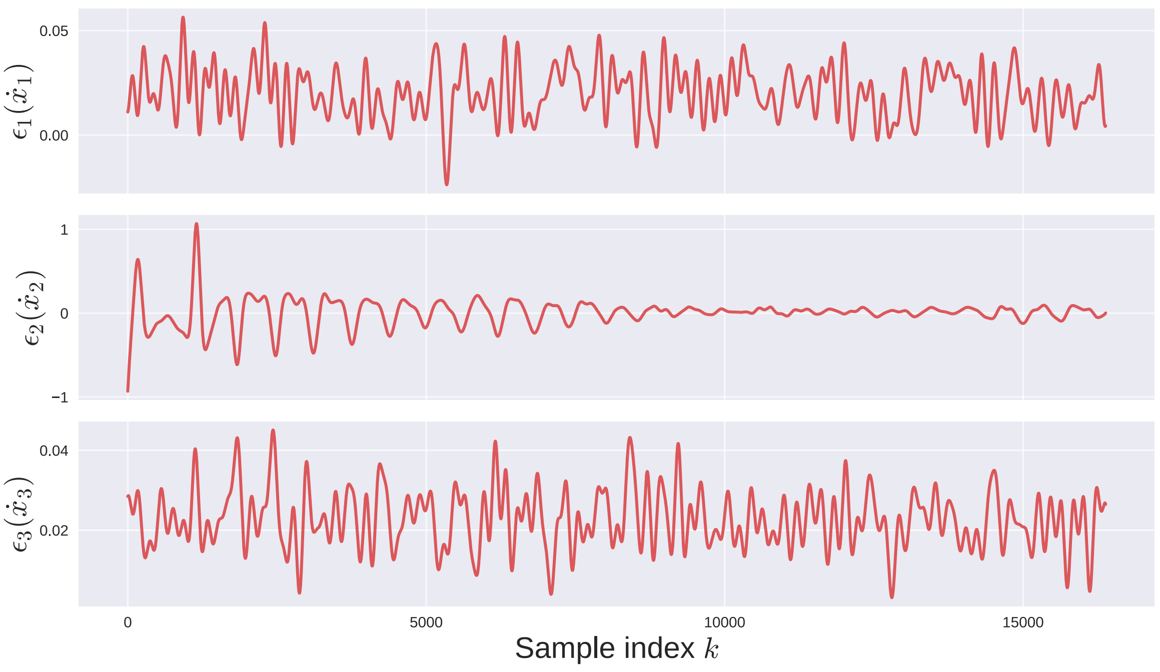
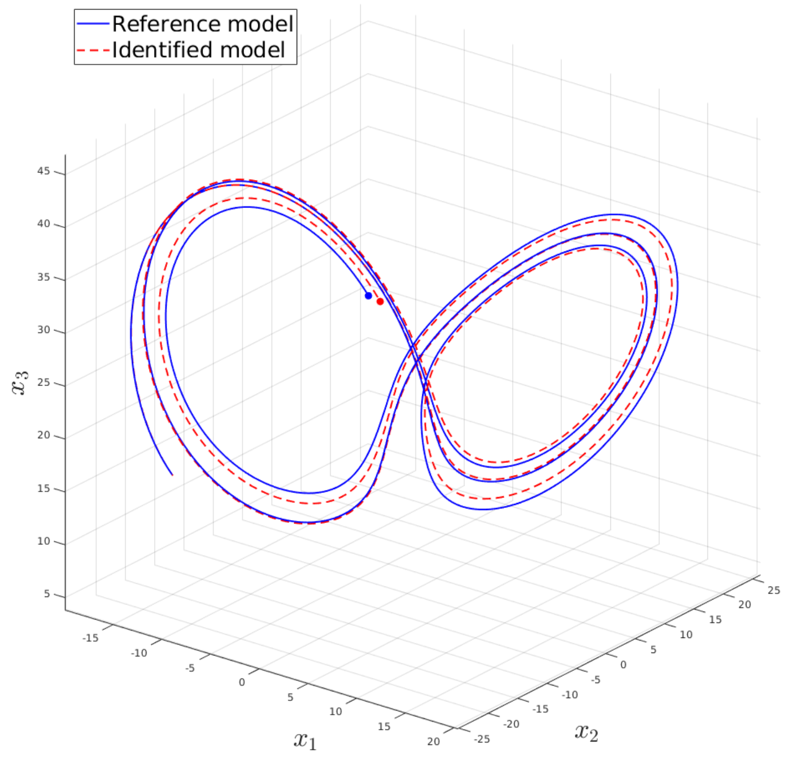


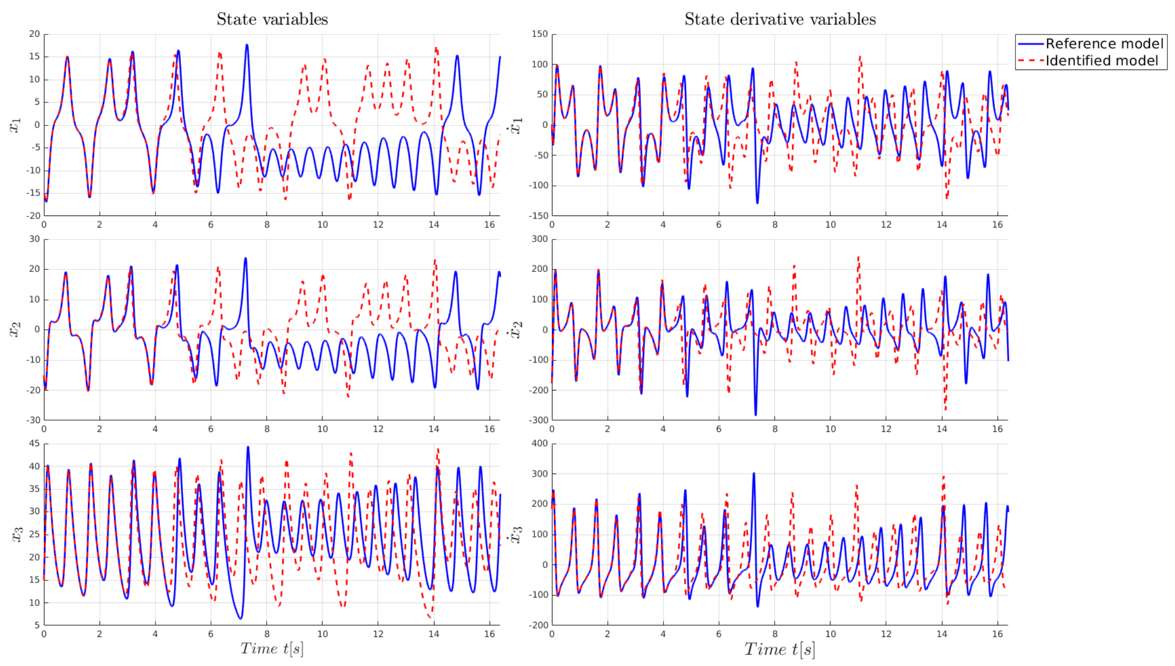
Fig. 30: State derivative prediction errors ϵ_i.

It is expected that error metrics calculated using test data should always be worse than the same metrics calculated using testing data. In this case, the reason they're lower is because the testing metrics were calculated from a clean validation dataset, while the training metrics were calculated from the dataset with filtered measurements and estimated derivatives. Most of the training RMSE comes from the inaccuracies in derivative estimations; the data was wrong, not the model derived from them.

Another way to test the model is using a full simulation. First, I use the discovered models to generate a MATLAB function using SymPy's code generator. The generated function has to be slightly edited for compatibility with MATLAB's `ode45` solver. Then I generate a new random noise process for the input, set an initial state and simulate both the real and the identified system.



(a) The first 4 seconds of the state trajectories.



(b) The full state and state derivative trajectories

Fig. 30: Results of a simulation of both the real and the identified system from the same initial state. The full visualization including the $3D$ trajectory is at: https://git.io/JBPQs

Both trajectories are close to each other at the start. Then, as the small errors in derivative predictions accumulate, the trajectories eventually decouple. Lorenz system is the most known example of a chaotic system - a system where small differences, either in the initial condition or in the parameters, lead to vastly different results. Chaotic systems are in reality quite common, as will be apparent in the next section. The consequence of chaos is that even if we have a very accurate model, we cannot use it to predict the real state arbitrarily far into the future. The further in time the prediction, the less certain it is.

8.6 Discussion

The method successfully discovered an accurate model even from data corrupted by noise. The identified model's ability to estimate the state derivatives $\dot{x}_i(\mathbf{x}, \mathbf{u})$ given $\mathbf{x}$ and $\mathbf{u}$ was validated using a validation data set generated by a separate simulation. The validation error was lower than the training error, because most of the training error was caused by inaccuracies in the training data itself. The identified model was then also validated by a simulation. The identified model's state trajectory was close to the real trajectory, but as a consequence of the chaotic nature of the Lorentz system, it eventually decoupled. Qualitatively, the identified model's trajectory was indistinguishable from the real trajectory. The Lorenz system was great for demonstrating many concepts that will be important in the next sections.

9 Model identification of a simulated pendulum-cart system

9.1 First-principles model derivation

For the purposes of generating training data and informing ourselves about the structure of the true dynamics of the system, let's start by deriving the analytical, first-principle model of the system using Euler-Lagrangian mechanics, following the procedure detailed in [21]. A pendulum-cart system has two degrees of freedom, the linear displacement of the cart s and the angle of the mounted pendulum φ. This implies that the true system consists of two second order differential equations describing the linear acceleration of the cart $\ddot{s}$ and the angular acceleration of the pendulum $\ddot{\varphi}$. The pendulum's 0 angle is at the stable equilibrium, where the pendulum is down. The angle coordinates use the right-hand rule, where the angle grows in the counter-clockwise direction.
First, let's define the Lagrangian as:

$$L = T - V \tag{60}$$

where T and V stand for the kinetic and the potential energy respectively. The kinetic energy T is then defined as:

$$T = \frac{1}{2} m_p \dot{\mathbf{P}}_p^{\mathrm{T}} \dot{\mathbf{P}}_p + \frac{1}{2} I_p \dot{\varphi}^2 + \frac{1}{2} m_c \dot{s}^2 \tag{61}$$

where m_p, m_c are the masses of the pendulum and the cart, I_p is the moment of inertia of the pendulum with respect to its center of mass, and $\mathbf{P}_p$ is the vector of x-y coordinates of the pendulum's center of mass.
The potential energy of the system V, assuming zero potential at the y coordinate of the cart, is simply the potential energy of the pendulum:

$$V = g m_p P_{p,y} \tag{62}$$

where $P_{p,y}$ is the y coordinate of the pendulum's center of mass.
To incorporate velocity-proportional friction forces into the model, let's define a Rayleigh dissipation function.

$$R = \frac{1}{2} b_p \dot{\varphi}^2 + \frac{1}{2} b_c \dot{s}^2 \tag{63}$$

where b_p and b_c are the damping coefficients of the pendulum joint and of the linear drive respectively. The friction forces are proportional to the first power of the respective velocity.

For generality, let's define the vector of generalized coordinates $\mathbf{q}$ as:

$$\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} s \\ \varphi \end{bmatrix} \tag{64}$$

The vector of generalized forces is fortunately quite simple:

$$\mathbf{Q} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} = \begin{bmatrix} u \\ 0 \end{bmatrix} \tag{65}$$

Where u is the force acting on the cart in the direction of q_1. The force u is generated by the linear drive, whose internal dynamics are omitted, because the circuit dynamics are much faster than the dynamics of the mechanical system and the simplification therefore doesn't significantly decrease the model's accuracy. The vector of generalized forces should contain all non-conservative forces, including the velocity-dependent friction forces like those defined by the Rayleigh dissipation function. However, in this thesis, I'll move the Rayleigh dissipation term to the left-hand side of the equation.

Finally, let's use the Euler-Lagrange equations to derive the equations of motion of the system.

$$\frac{d}{dt}\frac{dL}{d\dot{q}_k} - \frac{dL}{dq_k} + \frac{dR}{d\dot{q}_k} = Q_k \qquad k = 1, 2 \tag{66}$$

This procedure returns two implicit equations of motion, from which we can derive the explicit formulations for $\ddot{s}$ and $\ddot{\varphi}$. The equations of motion from (66) contain second order derivatives, each equation contains both second order terms. Such equations are difficult to re-order algebraically by hand, especially given their size, but symbolic math software packages, such as MATLAB's symbolic math toolbox, can greatly simplify the process. It is convenient to transform a system of two second order ODEs into a system of four first order ODEs. This procedure is closely connected with the concept of a state-space.

9.2 State-space representation

The system of two second order ODEs describing the cart-pendulum dynamics can be decomposed into a system of four first order ODEs in the following way:

$$\mathbf{x} = \begin{bmatrix} \mathrm{x}_1(t) \\ \mathrm{x}_2(t) \\ \mathrm{x}_3(t) \\ \mathrm{x}_4(t) \end{bmatrix} = \begin{bmatrix} \mathrm{s(t)} \\ \phi\mathrm{(t)} \\ \dot{\mathrm{s}}\mathrm{(t)} \\ \dot{\phi}\mathrm{(t)} \end{bmatrix} \implies \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x} = \dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathrm{x}}_1(t) \\ \dot{\mathrm{x}}_2(t) \\ \dot{\mathrm{x}}_3(t) \\ \dot{\mathrm{x}}_4(t) \end{bmatrix} = \begin{bmatrix} \dot{\mathrm{s}}\mathrm{(t)} \\ \dot{\phi}\mathrm{(t)} \\ \ddot{\mathrm{s}}\mathrm{(t)} \\ \ddot{\phi}\mathrm{(t)} \end{bmatrix} \tag{67}$$

The full state vector $\mathbf{x}$ consists of the generalized coordinates (x_1 and x_2) and generalized velocities (x_3 and x_4). Notice that the state variables x_3 and x_4 are equal to the state derivative variables $\dot{x}_1$ and $\dot{x}_2$, and that only the state derivative variables $\dot{x}_3$ and $\dot{x}_4$, or the accelerations, are unknown.

Doing this decomposition by hand can often be very time-consuming and algebraically non-trivial, especially as the dimension of the state space increases. Fortunately, there are symbolic solvers which can do this automatically. I do the entire sequence of analytical model derivation steps using MATLAB and its symbolic math toolbox. I will describe only the most important and non-trivial parts of the code in this thesis. The full code is available in the attachments.

The differential order reduction step is done using the `reduceDifferentialOrder(eqs, vars)` function, which takes in the vector of higher order symbolic ODEs `eqs` and a vector of symbolic variables (in this case, generalized coordinates) `vars`. It returns a vector of first order symbolic ODEs `VF` (Vector Field) and a vector of state variables `state_vars`. These can be reordered into the mass-matrix form $\mathbf{M}\dot{\mathbf{x}} = \mathbf{F}$ using the function `massMatrixForm(VF, state_vars)`. Since the state-space model is described only by the state derivative vector $\dot{\mathbf{x}}$, we must solve for it as:

$$\dot{\mathbf{x}} = \mathbf{M}^{-1}\mathbf{F}$$

which can be done using MATLAB's backslash operator:

$$\dot{\mathbf{x}} = \mathbf{M}\backslash\mathbf{F}$$

Then, after some relatively trivial substitutions, we get the full nonlinear state-space model of the system. The first two state derivative variables, $\dot{x}_1 = x_3$ and $\dot{x}_2 = x_4$ stand for the cart's linear velocity and the pendulum's angular velocity respectively. The other two state derivative variables, $\dot{x}_3$ and $\dot{x}_4$, stand for the cart's acceleration and the pendulum's angular acceleration. The system dynamics are fully described by the accelerations as:

$$\dot{x}_3 = \frac{A(\mathbf{x},u)}{B(\mathbf{x},u)} \tag{68a}$$

$$\begin{aligned} A(\mathbf{x},u) = I_1\,u(t) + {a_1}^2\,m_1\,u(t) - I_1\,b_c\,x_3(t) + +{a_1}^3\,{m_1}^2\,\sin(x_2(t))\,x_4^2(t) + \\ -{a_1}^2\,b_c\,m_1\,x_3(t) + +a_1\,b_1\,m_1\,\cos(x_2(t))\,x_4(t) + +{a_1}^2\,g\,{m_1}^2\,\cos(x_2(t))\,\sin(x_2(t)) + \\ + I_1\,a_1\,m_1\,\sin(x_2(t))\,x_4^2(t) \end{aligned} \tag{68b}$$

$$B(\mathbf{x},u) = -{a_1}^2\,{m_1}^2\cos^2(x_2(t)) + +{a_1}^2\,{m_1}^2 + m_c\,{a_1}^2\,m_1 + I_1\,m_1 + I_1\,m_c \tag{68c}$$

$$\dot{x}_4 = \frac{C(\mathbf{x},u)}{D(\mathbf{x},u)} \tag{69a}$$

$$\begin{aligned} C(\mathbf{x},u) = b_1\,m_1\,x_4(t) + b_1\,m_c\,x_4(t) + +a_1\,g\,{m_1}^2\,\sin(x_2(t)) + a_1\,m_1\,\cos(x_2(t))\,u(t) + \\ + a_1\,g\,m_1\,m_c\,\sin(x_2(t)) + +{a_1}^2\,{m_1}^2\,\cos(x_2(t))\,\sin(x_2(t))\,x_4^2(t) + \\ - a_1\,b_c\,m_1\,\cos(x_2(t))\,x_3(t) \end{aligned} \tag{69b}$$

$$D(\mathbf{x}, u) = -{a_1}^2\,{m_1}^2\cos^2(x_2(t)) + +{a_1}^2\,{m_1}^2 + m_c\,{a_1}^2\,m_1 + I_1\,m_1 + I_1\,m_c \tag{69c}$$

Notice that the acceleration equations are rational.

9.3 Simulation

Now that we have the full nonlinear state-space model

$$\dot{\mathbf{x}}(\mathbf{x}, u) = \begin{bmatrix} \dot{x}_1(\mathbf{x}, u) \\ \dot{x}_2(\mathbf{x}, u) \\ \dot{x}_3(\mathbf{x}, u) \\ \dot{x}_4(\mathbf{x}, u) \end{bmatrix}$$

we can generate a MATLAB ODE function from the symbolic equations and simulate the system using the `ode45` solver as before. First, we must define the physical parameters of the system.

Table 3: The physical parameters of the simulated pendulum-cart system in base units.

Physical parameter	Meaning	Value
I_1	Pendulum's moment of inertia around its center of mass	$0.0227\,\mathrm{kg\,m^2}$
a_1	Distance from the pendulum joint to its center of mass	$0.18\,\mathrm{m}$
b_c	Linear cart friction coefficient	$10\,\frac{\mathrm{N\,s}}{\mathrm{m}}$
b_1	Pendulum joint friction coefficient	$0.15\,\frac{\mathrm{N\,s}}{\mathrm{m}}$
m_c	Mass of the cart	$0.8\,\mathrm{kg}$
m_1	Mass of the pendulum	$1\,\mathrm{kg}$
g	Gravitational field intensity	$9.81\,\frac{\mathrm{N}}{\mathrm{kg}}$

By defining an initial state, we can simulate the system's dynamics and measure the systems response. For data collection, this however isn't ideal, as the system would eventually converge to some stable state and stop. To collect better measurements, the system input u will have to continually add energy to the system. Possible functions for $u(t)$ could be for example trigonometric functions, such as $\sin(t)$. The problem with defining $u(t)$ as a simple trigonometric function of time is that the signal is very sparse in the frequency domain and doesn't reveal that much information about the system, because the generated state trajectory will likely be stuck in some cyclical state trajectory. Naturally, the best choice for the forcing signal is some band-limited random process. Earlier, the random noise process's purpose was to de-correlate the feedback-looped inputs from the states. In this case, the random noise process will have an additional use. It will be used to generate a reference trajectory for the cart's position x_1.

First, I generate a band-limited noise process, just as before. Then I generate a shifted square wave function, with maximal value 1 and minimal 0, and multiply the band-limited noise signal by the square wave. This way, the cart trajectory will periodically return to 0. Because the signal becomes discontinuous, and we can't expect the cart to move instantaneously, the signal is filtered again. Then I normalize the signal by multiplying it with its maximal absolute value. This way, the maximal deviation from the origin will be 1 in either direction. Then I multiply the signal by the maximal allowed deviation from the origin - in this case 0.5 meters. The entire sequence of trajectory generation is shown in the Figure 31

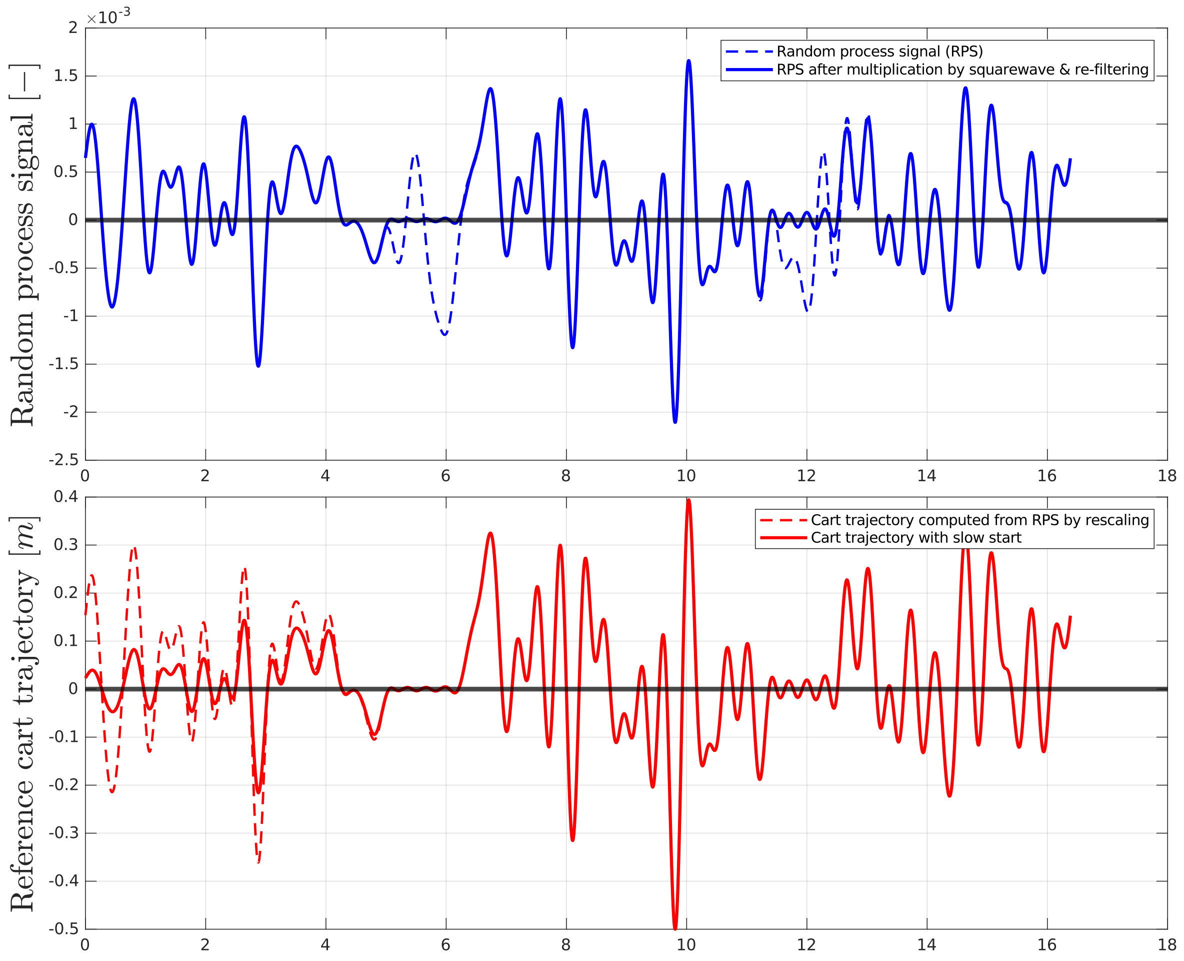


Fig. 31: The sequence of steps for computing a cart trajectory for data collection.

The cart trajectory, let's call it $w_1(t)$, prescribes the desired trajectory of the x_1 state variable during the simulation. Our only input to the system is the force $u(t)$ acting on the cart. Assuming that the force $u(t)$ is directly proportional to the cart's acceleration, we can get the shape of the signal $u(t)$ by double-differentiating the cart trajectory $w_1(t)$. The amplitudes of the $u(t)$ signal are uncertain, calculating the signal exactly would require already knowing the system parameters. But we can keep multiplying the $u(t)$ trajectory and observing the simulation results; if the cart moves too much, make the signal weaker; if it doesn't move enough, make it stronger. Alternatively, the linear drive could have a control system with a reference acceleration as an input directly.

Because the signal $u(t)$ was generated as a discrete function, it must be transformed into a continuous function for the purpose of simulation using a variable step solver. This is done, as before, by linear interpolation. The result is a continuous time-dependent function $u(t)$ that can be embedded into the system of equations $\dot{\mathbf{x}}$ for simulation.

I use the adaptive Runge-Kutta 45 time-stepping scheme for numerical simulation of

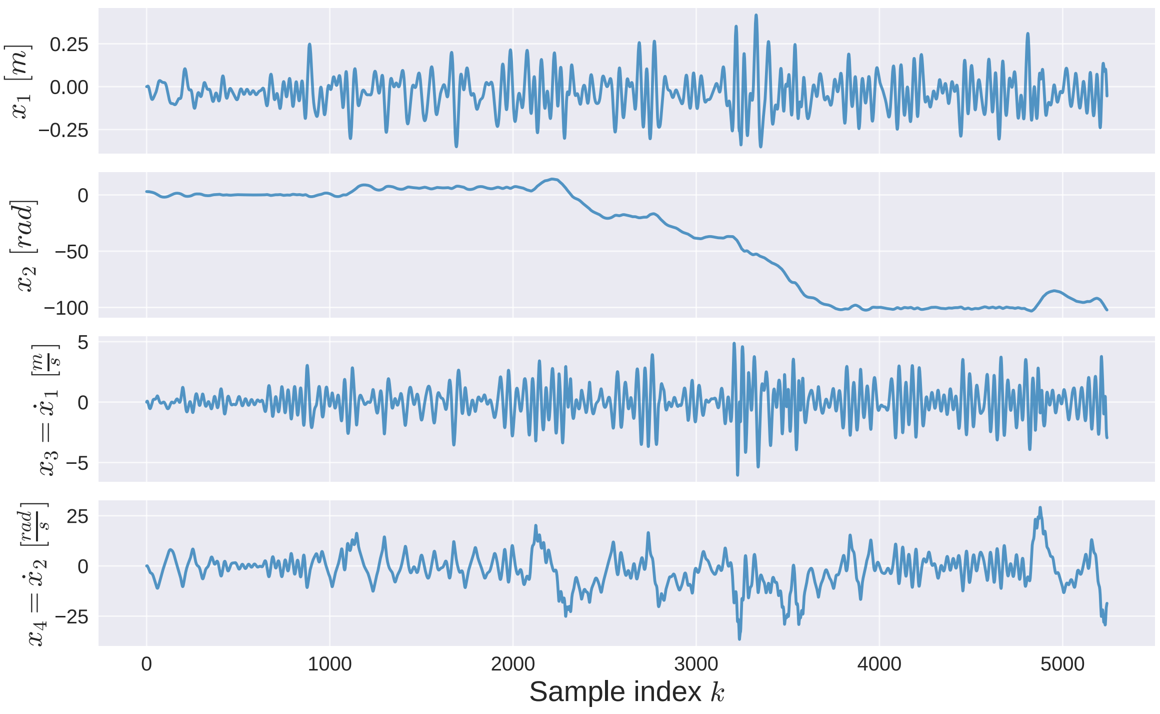


Fig. 32: The state trajectories $\mathbf{X}$ from the pendulum-cart system simulations.

the system, which is implemented in MATLAB as `ode45(odefun, tspan, x0)`, where `odefun` is the system of ODEs $\dot{\mathbf{x}}$, `tspan` is the time span of the simulation, and `x0` is the initial state. The function can return either pairs `[t,x]` (time and state vector), or a solution object `sol`, which is more convenient, because it can be used to return the state vector values at a constant sampling period.

The output of the simulation is a state trajectory $\mathbf{x}(t)$. Realistically, we'd only be measuring two state variables, namely the cart position $\mathbf{x}_1(t)$ and the pendulum angle $\mathbf{x}_2(t)$. The other two state variables are velocities and can be obtained from $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$ by numerical differentiation. The state derivative trajectory $\dot{\mathbf{x}}(t)$ is again four-dimensional, where the first two variables are the previously computed velocities and the second two are accelerations, which are computed by again numerically differentiating the velocity variables.

9.4 Function library

After obtaining the full state trajectory $\mathbf{x}(t)$ and the state derivative trajectory $\dot{\mathbf{x}}(t)$, the next step is creating the function library. This is a critical part of the identification procedure, because to create an accurate model, the function library must contain all the terms in the actual implicit dynamics. To make an educated guess about the active terms, some expert knowledge about the modeled system is necessary. The library can contain terms that are inactive in the *real* model, because their respective parameters will be set to zero by the sparse regression algorithm. However, the number of terms in the library cannot be increased arbitrarily, mostly because large function libraries

increase the regression's sensitivity to noise.
Since we've derived the analytical model (68, 69) before, we can see which terms in the implicit dynamics are truly active. We could create the function library only from the terms in the analytical solution, this would however make sparse regression pointless, as there wouldn't be any parameters to zero out. It'd also mean that we already know the complete real model structure, and the regression task would become a relatively simple parameter estimation by least-squares regression.
It's reasonable to make a guess that the cart's linear velocity and the pendulum's angular velocity will be present in the dynamics. The force on the cart u is also obviously affecting the dynamics. Due to the *rotational* nature of the pendulum, we can also assume that trigonometric functions of the pendulum's angle will play an important role. Rotating masses imply centripetal accelerations, which are a function of the second power of angular velocity. Joint frictions are typically a function of the difference in rotation velocity, but in this case, we only have one rotating part, so the difference is simply the pendulum's rotational velocity.
This narrows down the space of candidate terms to different combinations of the terms $x_3(t)$, $x_4(t)$, $\sin(x_2(t))$, $\cos(x_2(t))$, $u(t)$, the constant term 1, and our target variables, which are the accelerations $\dot{x}_3$ and $\dot{x}_4$. I think of these as the *basis* terms. All library terms are created as a combination of the basis terms. To make the notation more concise and to stress the mutual independence of the basis terms, let's collect and rename them as follows:

$$\{x_3, x_4, \sin(x_2), \cos(x_2), u, 1, \dot{x}_3, \dot{x}_4\} = \{y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8\} = \mathbf{Y} \tag{70}$$

We can now create the function library as an informed combination of the basis terms y_i. From the analytically derived model ((68, 69)), we can see that the highest order term is $\cos(x_2)\sin(x_2)x_4^2$, or, using our new notation, $y_4 y_3 y_2^2$. Let's use this information to create all possible 4th-order terms from the set of basis terms y_i. Note that not all candidate functions will *actually* be of 4th-order, because the term y_6 is an identity. A 4th-order monomial $y_1 y_6^3$ is actually just the 1-st order monomial x_3. The resulting function library is nearly identical [2] as if we excluded the identity term y_6 from the function space and simply created the function library as a union of all sets of monomials of y_i *up* to the 4th order.
The set of monomials is created as a set of all possible multi-combinations (combinations with replacement) of size 4 from the set $\mathbf{Y}$, which has 8 elements. The number of terms is given by the formula:

$${}^{d}C_n = \frac{(d+n-1)!}{n!(d-1)!} \tag{71}$$

Where d is the number of elements in $\mathbf{Y}$ and n is the monomial order. We have 8 basis terms and 4th monomial order, therefore we have 330 candidate functions in the function library. This is a lot of candidate functions, but some of them can be dropped according to some problem-specific knowledge.

[2] It'd just be missing the constant function $1(t)$

Alternatively, instead of creating a huge set of candidate functions and then reducing it, the candidate functions could all be picked manually. The issue with this approach is that if even just one important candidate function is missing from the set during regression, the correct model won't be found. On the other hand, if the function library contains functions that are irrelevant, they'll simply be discarded by the sparsity-promoting regression algorithm. Missing an important candidate function is much more expensive than including irrelevant ones, therefore it's safer to create too many candidate functions and then manually discard those that go against the expert opinion.
I removed all candidate functions that met at least one of these conditions:

- The candidate function contains a basis term with a power higher than 2. The highest expected single basis term order is 2, for example, the square of the angular velocity, x_4^2 is allowed (implies centripetal acceleration), but the cube of angular velocity, x_4^3 isn't.
- The candidate function contains more than 3 unique terms. Candidate functions with a rich mix of basis terms, for example $x_3 x_4 \sin(x_2) u$, are not expected. This implies that a 4th-order candidate function must contain at least one squared basis term.
- The candidate function contains a combination of the terms x_3, x_4, u, $\dot{x}_3$ or $\dot{x}_4$. For example, a function such as $x_3 u \dot{x}_3 \dot{x}_4$ contains both accelerations, which isn't acceptable in our state-space formulation. This means that those terms can only appear with one or more of the trigonometric basis terms.
- The candidate function contains $\sin(x_2)^2$. This is because of the identity $\sin(x_2)^2 + \cos(x_2)^2 = 1$, and because both $\cos(x_2)^2$ and 1 are already in the function library. Including functions with $\sin(x_2)$ would create ambiguity and would result in poorly conditioned regression.

This reduces the number of viable candidate functions to 94. For comparison, the analytically derived model (68, 69) contains only 13 terms in total.
This function library contains candidate functions for identifying both $\dot{x}_3$ and $\dot{x}_4$. For each of these target variables, the function library is further reduced so that it doesn't contain the other target variable(s). For example, when identifying the model for $\dot{x}_3$, all the candidate functions containing $\dot{x}_4$ will be dropped from the library, further reducing its total number of candidate functions to 35.

The correlation matrix of the function library for identifying the implicit ODE for $\dot{x}_3$ is shown in Figure 33. When reducing the set of candidate functions, it's a good idea to check for highly correlated functions. Ideally, we want the correlation matrix to be as diagonally dominant as possible, because correlated functions make the regression problem ill-posed. If two functions are strongly correlated, we can investigate them further. One such correlation appears for example when $\sin {x_1}^2$ and $\cos {x_1}^2$ both appear in the library; in this case, they form a duality due to the equality $\sin(x)^2 + \cos(x)^2 = 1$ and one of them can be discarded.

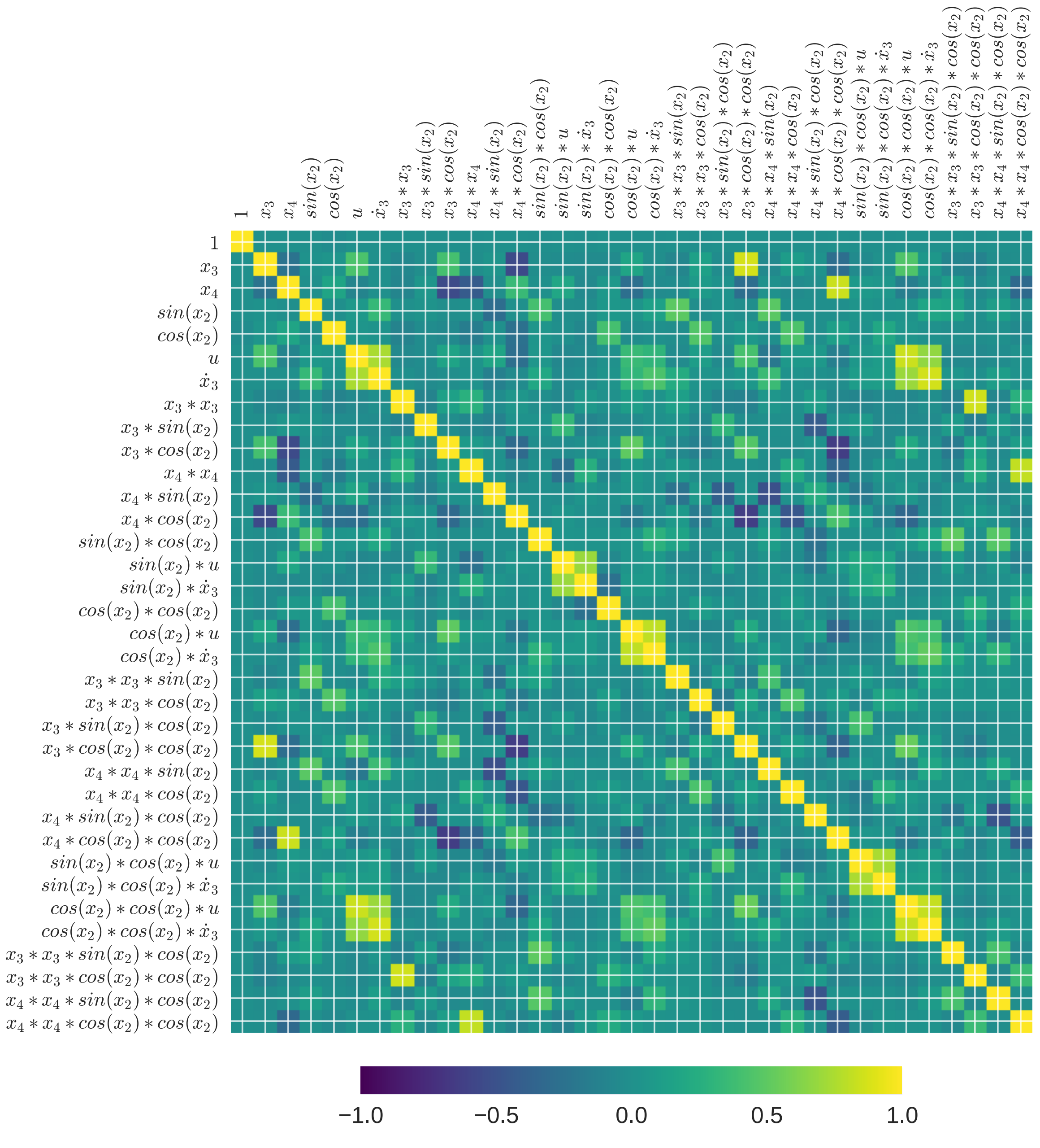
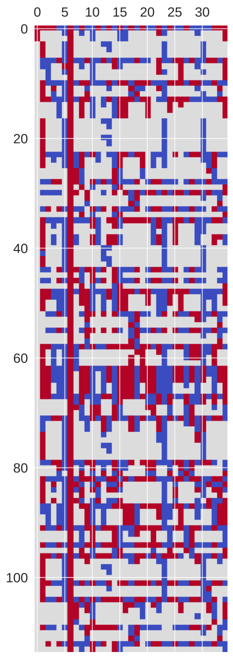


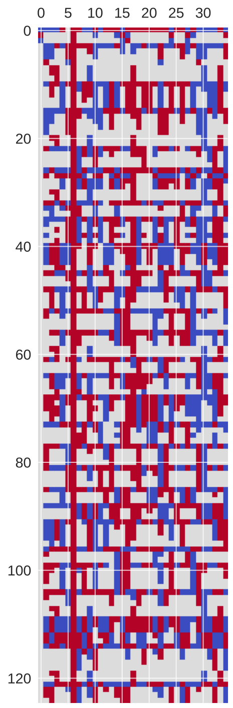
Fig. 33: Correlation matrix of the function library for identifying the implicit ODE describing $\dot{x}_3$, the cart acceleration.

9.5 Identification using clean data

First, let's do the identification with clean data, to check if the function library $\mathbf{\Theta}$ contains all needed terms. The dynamics we wish to identify are now rational. The method relies on linear regression, so the dynamics must be expressible as a linear combination of the candidate functions. A rational model can be rewritten into a linear combination by multiplying the equations by their denominators and moving all terms to one side of the equation. The ODEs then become implicit. The SINDy-PI method described in 3.1.5 looks for implicit models by guessing that one of the candidate functions is in the implicit dynamics. If the guess is correct, then the correct model is likely to be identified. In the following examples, I ran the identification procedure by running the regression multiple times, using each candidate function θ_i from $\mathbf{\Theta}$ as the guess. For each candidate function θ_i, I generated 5 models, by varying the regression hyperparameters. With 35 candidate functions, that means a total of 175 implicit models was generated for both the cart acceleration $\dot{x}_3$ and the pendulum angular acceleration $\dot{x}_4$.



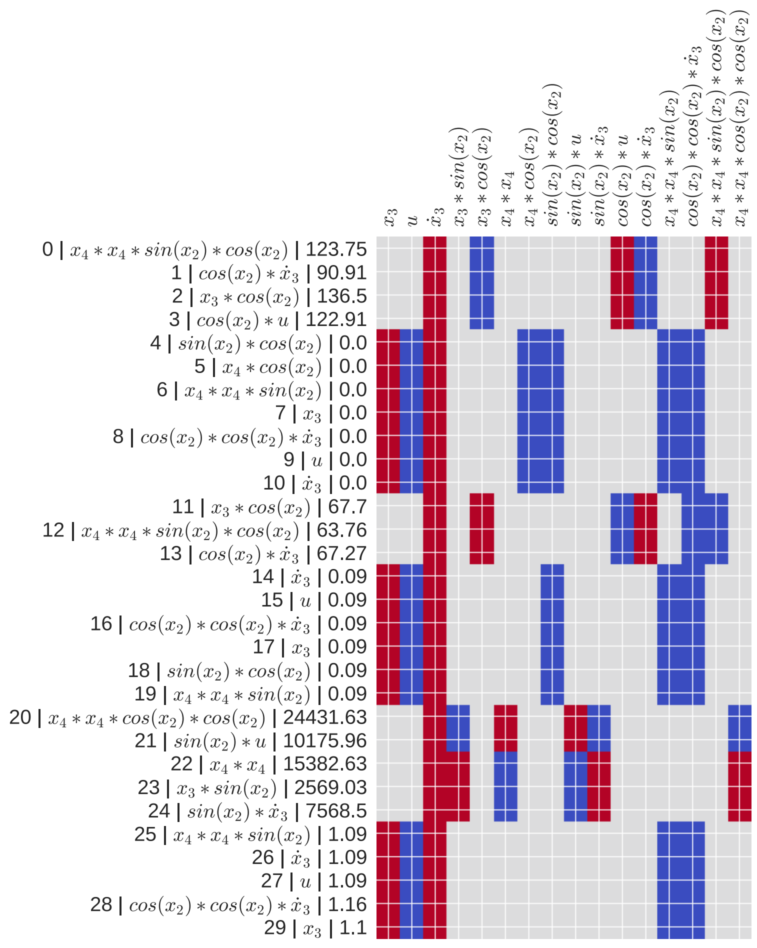
(a) All unique implicit models for $\dot{\mathbf{x}}_3$. trajectories.



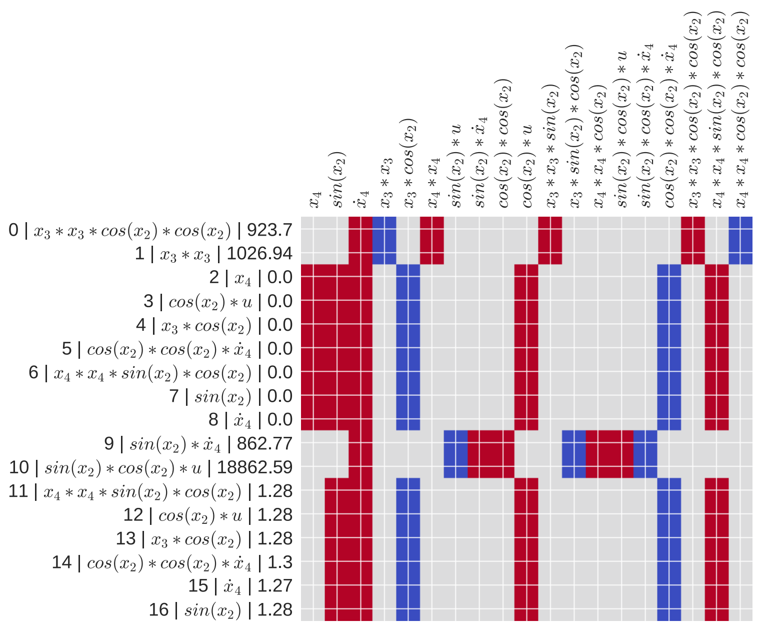
(b) All unique implicit models for $\dot{\mathbf{x}}_4$.

Fig. 34: Visualization of all unique identified implicit models. Each row is an implicit model and each column is a coefficient ξ_i of the candidate function θ_i. A red square symbolizes a positive coefficient value, a blue square a negative coefficient value.

To help us sift through the vast number of models, we can use the fact that if our function library $\mathbf{\Theta}$ contains all the functions needed to describe the *real* model equation, this model will be identified for many different right-hand side guesses of θ_i. Whenever the guess is *correct*, the correct model should be found. If we look for models that appear consistently in the full set of identified models, these consistent models are good candidates for further evaluation.



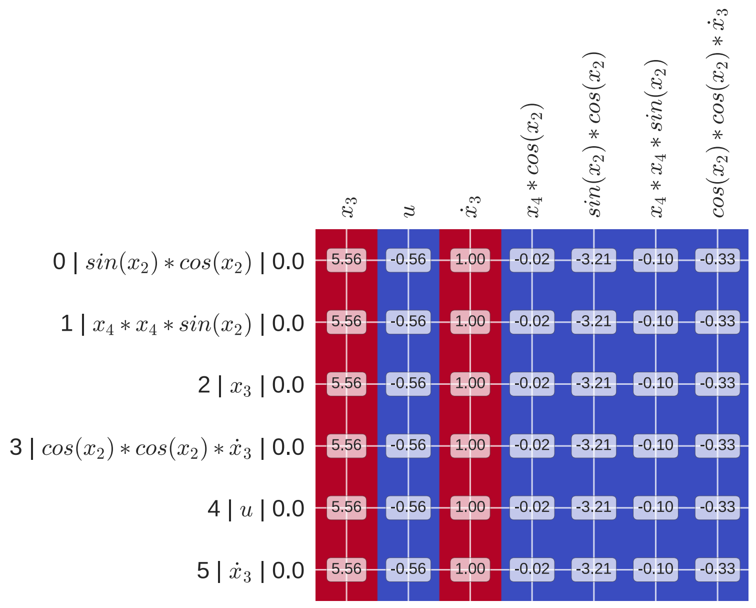
(a) Structurally consistent models for $\dot{\mathbf{x}}_3$. trajectories.



(b) Structurally consistent models for $\dot{\mathbf{x}}_4$.

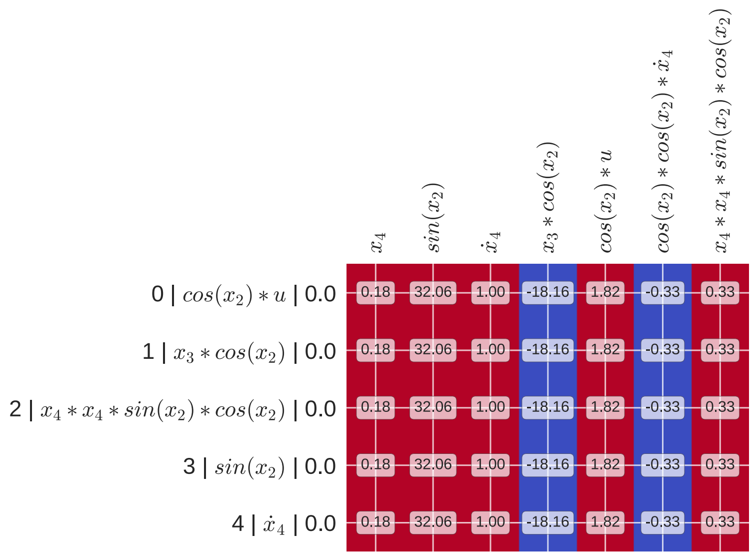
Fig. 35: Visualization of all structurally consistent, sparse implicit models identified from clean data.

The exact models with 0 training error were correctly identified from the data. In the next subsection, I'll repeat the identification using a more realistic dataset.



	x_3	u	$\dot{x}_3$	$x_4 * cos(x_2)$	$sin(x_2) * cos(x_2)$	$x_4 * x_4 * sin(x_2)$	$cos(x_2) * cos(x_2) * \dot{x}_3$
0 | $sin(x_2) * cos(x_2)$ | 0.0	5.56	-0.56	1.00	-0.02	-3.21	-0.10	-0.33
1 | $x_4 * x_4 * sin(x_2)$ | 0.0	5.56	-0.56	1.00	-0.02	-3.21	-0.10	-0.33
2 | x_3 | 0.0	5.56	-0.56	1.00	-0.02	-3.21	-0.10	-0.33
3 | $cos(x_2) * cos(x_2) * \dot{x}_3$ | 0.0	5.56	-0.56	1.00	-0.02	-3.21	-0.10	-0.33
4 | u | 0.0	5.56	-0.56	1.00	-0.02	-3.21	-0.10	-0.33
5 | $\dot{x}_3$ | 0.0	5.56	-0.56	1.00	-0.02	-3.21	-0.10	-0.33

(a) Exact models for $\dot{\mathbf{x}}_3$. trajectories.



	x_4	$sin(x_2)$	$\dot{x}_4$	$x_3 * cos(x_2)$	$cos(x_2) * u$	$cos(x_2) * cos(x_2) * \dot{x}_4$	$x_4 * x_4 * sin(x_2) * cos(x_2)$
0 | $cos(x_2) * u$ | 0.0	0.18	32.06	1.00	-18.16	1.82	-0.33	0.33
1 | $x_3 * cos(x_2)$ | 0.0	0.18	32.06	1.00	-18.16	1.82	-0.33	0.33
2 | $x_4 * x_4 * sin(x_2) * cos(x_2)$ | 0.0	0.18	32.06	1.00	-18.16	1.82	-0.33	0.33
3 | $sin(x_2)$ | 0.0	0.18	32.06	1.00	-18.16	1.82	-0.33	0.33
4 | $\dot{x}_4$ | 0.0	0.18	32.06	1.00	-18.16	1.82	-0.33	0.33

(b) Exact models for $\dot{\mathbf{x}}_4$.

Fig. 36: The exact models with 0 training RMSE, identified from clean data.

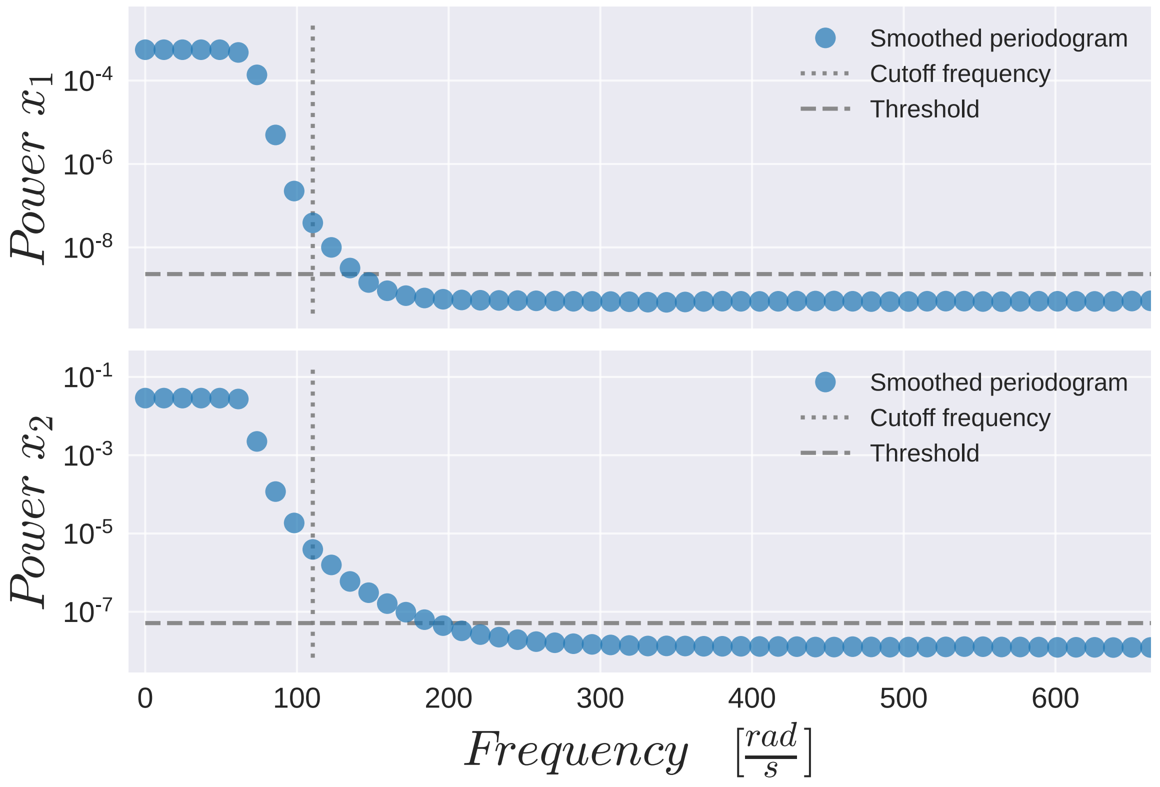
9.6 Identification using imperfect data

To simulate real-world conditions, I'll only use the measurements of the cart position x_1 and the pendulum angle x_2. White noise was added to both position and angular measurements, with a standard deviation of $0.0005\mathrm{m}$ and $0.0025\mathrm{rad}$ respectively. The measurements are filtered using a spectral filter. The cutoff frequencies are manually offset towards lower frequencies, so that the strong frequencies of the information signal are dominant in the estimated derivative signal. If the cutoff frequency was chosen as the frequency at which noise becomes stronger than signal, the noise frequencies close to the cutoff frequency would have a disproportionate effect on the derivative estimate.

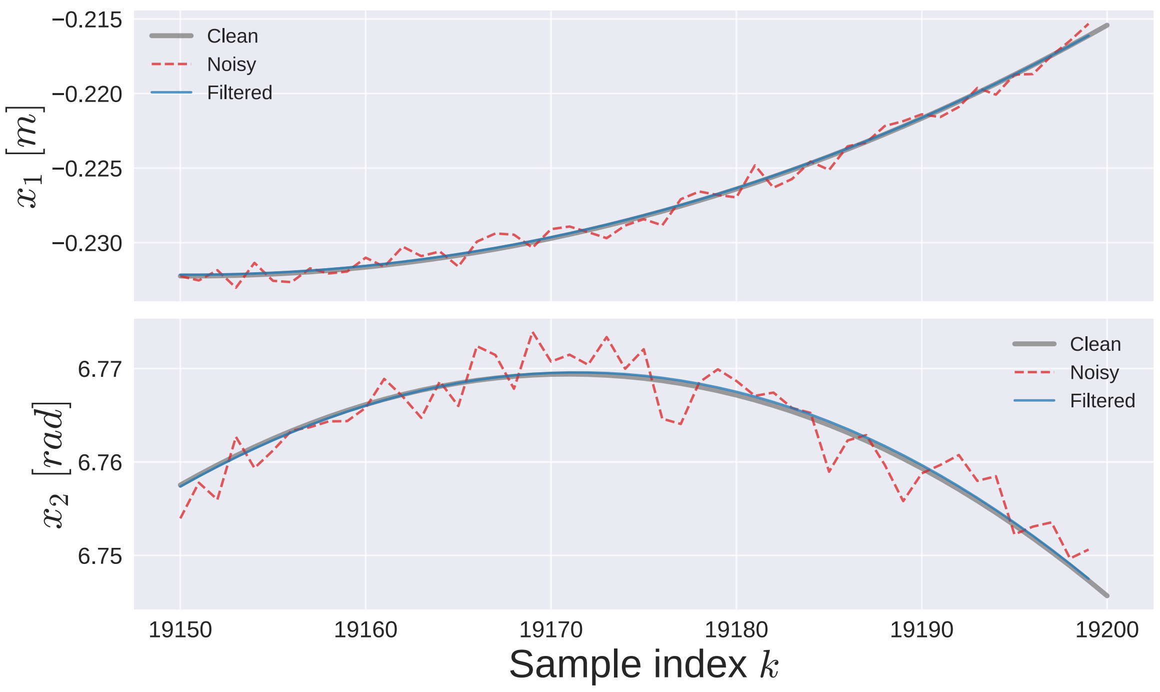
Next, the filtered measurements are used to estimate the cart velocity $\dot{x}_1 = x_3$ and the pendulum's angular velocity $\dot{x}_2 = x_4$. The derivative estimates are calculated using spectral differentiation. With these measurements and estimates, the state measurement matrix $\mathbf{X}$ is complete.

We want a model of the cart's acceleration $\dot{x}_3$ and the pendulum's angular acceleration $\dot{x}_4$. The respective acceleration measurement vectors are estimated from the (also estimated) velocities, again using spectral differentiation.

Now we have all the data we need to construct the candidate function library $\mathbf{\Theta}$ with the same structure as in the identification before. Again, the library is used to generate a set of models. The discovered unique and consistent models are shown in the next figure.



(a) Spectral filter cutoff frequencies for both measurement signals.



(b) Comparison of the clear, noisy and filtered signals.

Fig. 37: Plots showing the used filter settings and a part of the signal.

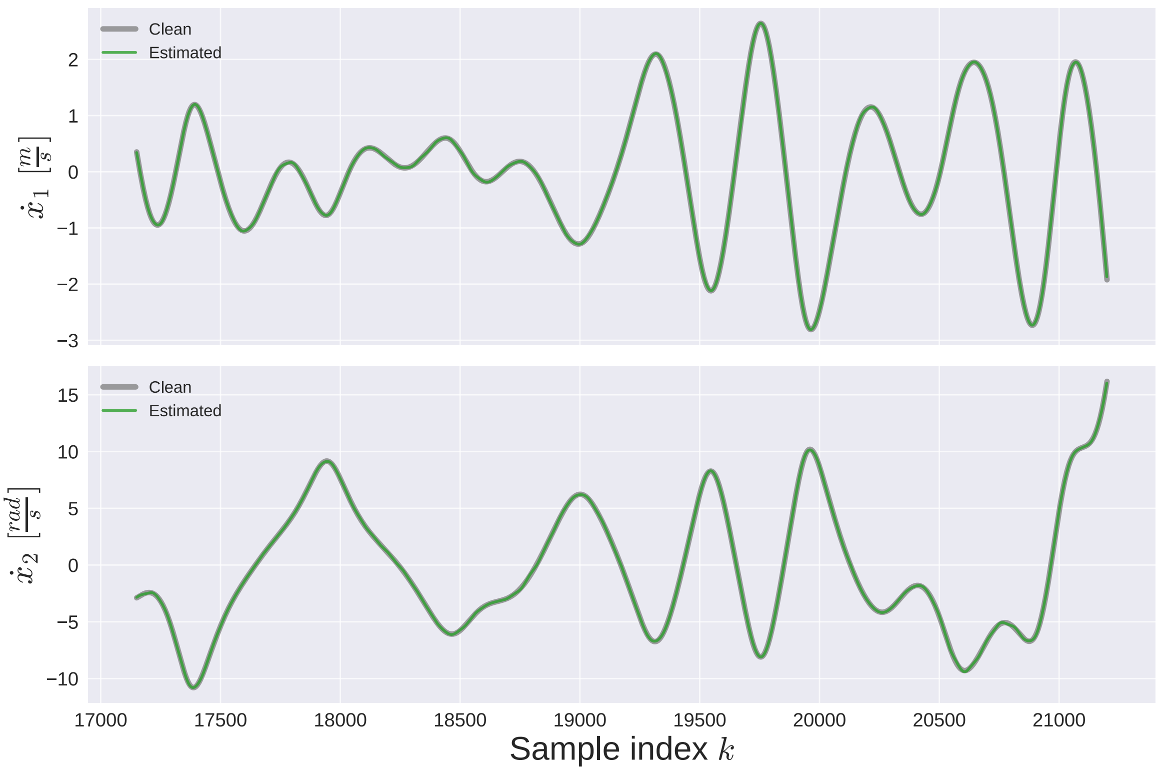


Fig. 38: Cart's linear velocity and the pendulum's angular velocity estimated using spectral differentiation.

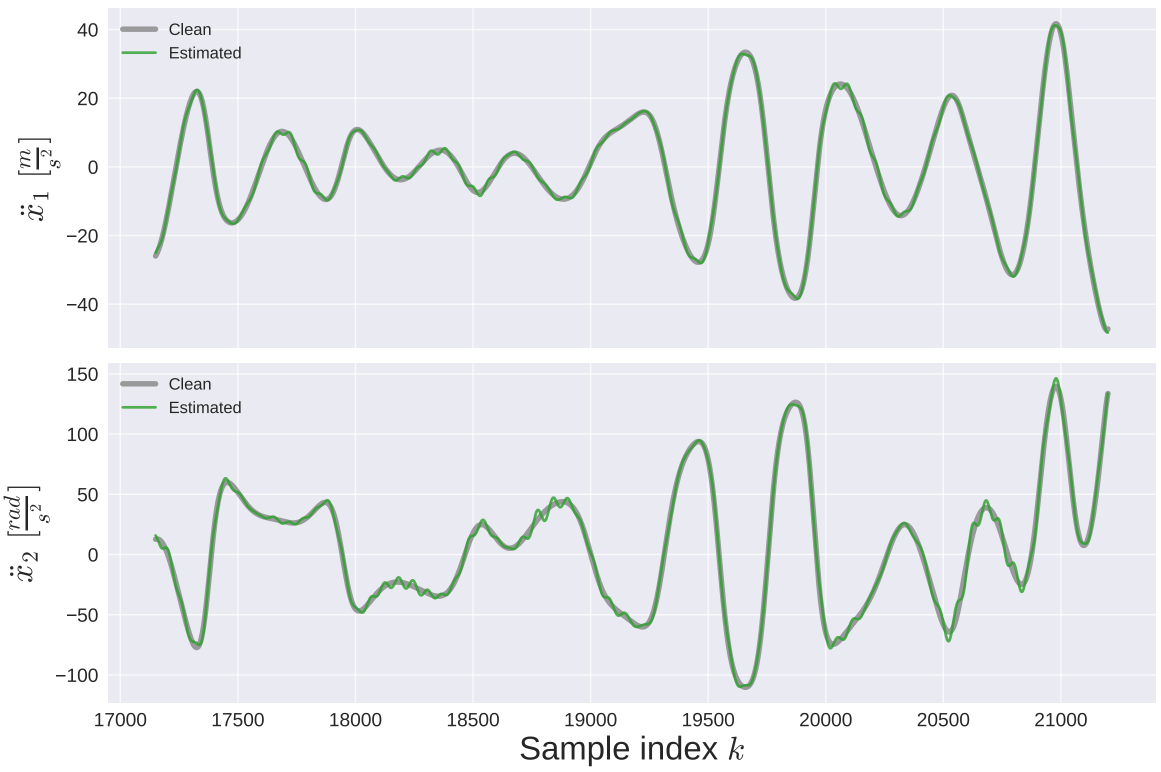
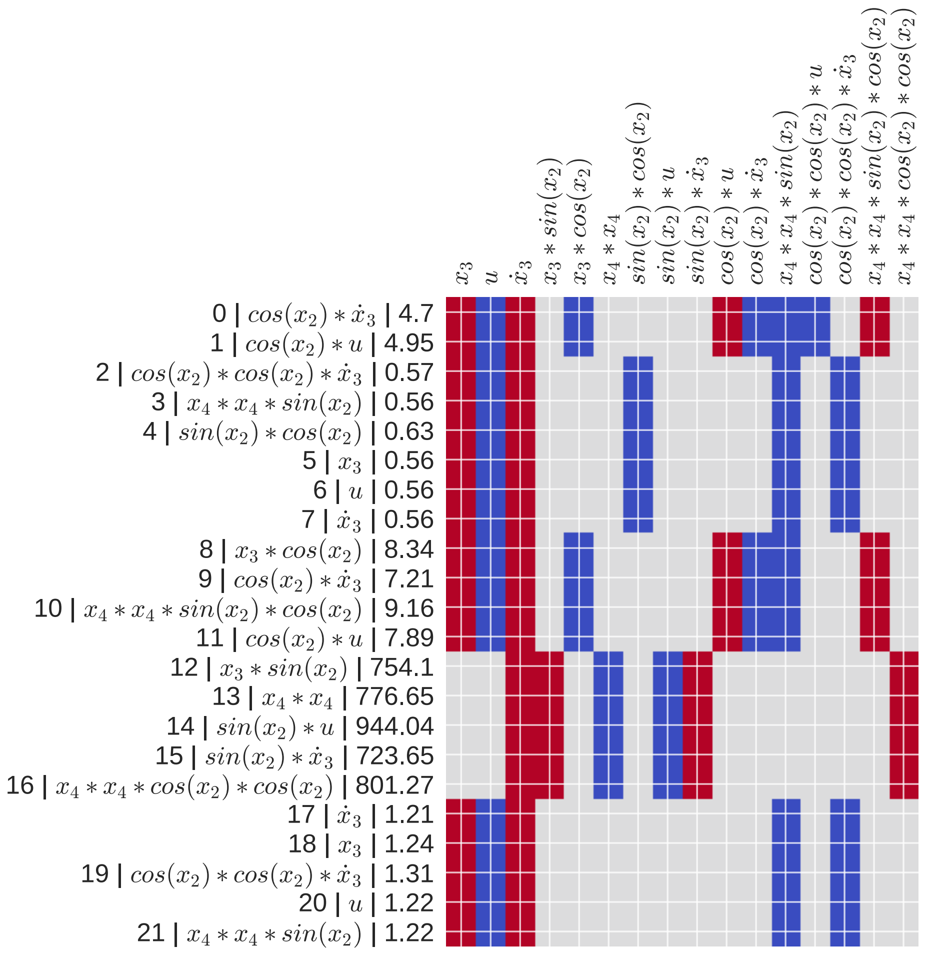
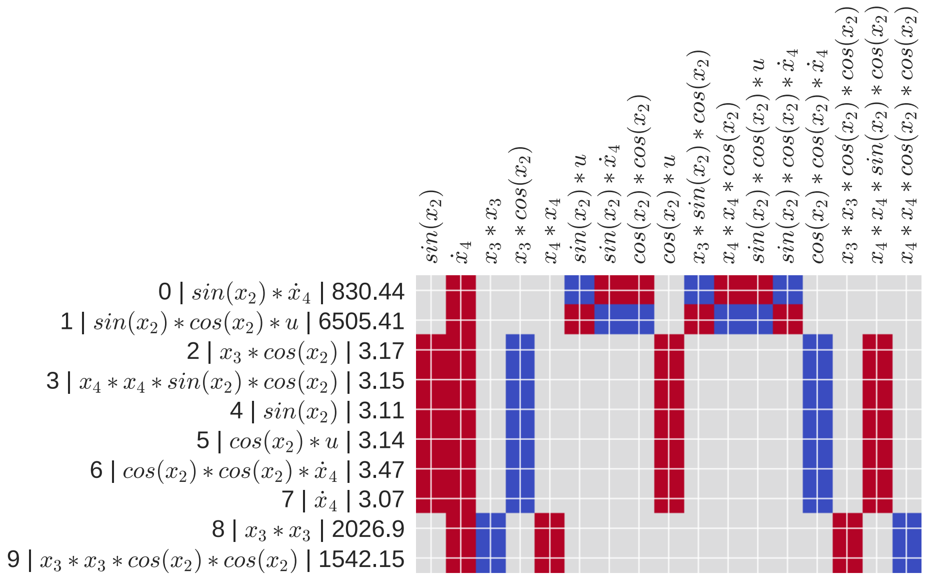


Fig. 39: Cart's linear acceleration and the pendulum's angular acceleration estimated using spectral differentiation.

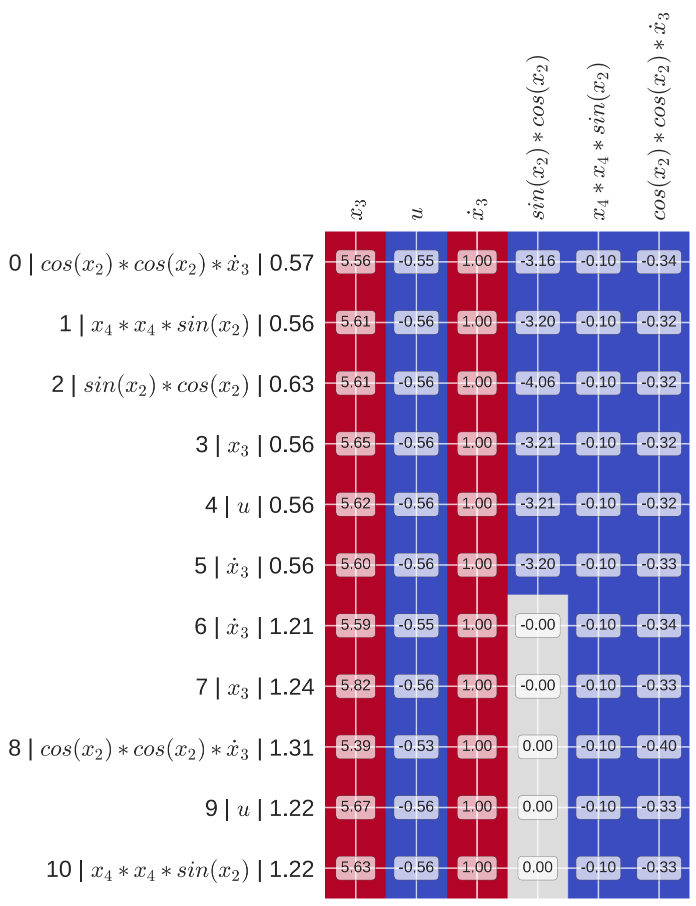


(a) Consistent models for $\dot{\mathbf{x}}_3$. trajectories.



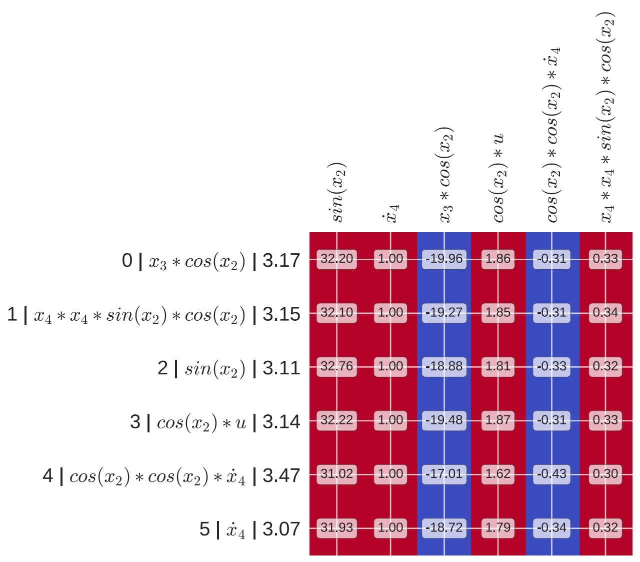
(b) Consistent models for $\dot{\mathbf{x}}_4$.

Fig. 40: Visualization of all structurally consistent implicit models identified using noisy measurements and estimated derivatives.



	x_3	u	$\dot{x}_3$	$sin(x_2) * cos(x_2)$	$x_4 * x_4 * sin(x_2)$	$cos(x_2) * cos(x_2) * \dot{x}_3$
0 | $cos(x_2) * cos(x_2) * \dot{x}_3$ | 0.57	5.56	-0.55	1.00	-3.16	-0.10	-0.34
1 | $x_4 * x_4 * sin(x_2)$ | 0.56	5.61	-0.56	1.00	-3.20	-0.10	-0.32
2 | $sin(x_2) * cos(x_2)$ | 0.63	5.61	-0.56	1.00	-4.06	-0.10	-0.32
3 | x_3 | 0.56	5.65	-0.56	1.00	-3.21	-0.10	-0.32
4 | u | 0.56	5.62	-0.56	1.00	-3.21	-0.10	-0.32
5 | $\dot{x}_3$ | 0.56	5.60	-0.56	1.00	-3.20	-0.10	-0.33
6 | $\dot{x}_3$ | 1.21	5.59	-0.55	1.00	-0.00	-0.10	-0.34
7 | x_3 | 1.24	5.82	-0.56	1.00	-0.00	-0.10	-0.33
8 | $cos(x_2) * cos(x_2) * \dot{x}_3$ | 1.31	5.39	-0.53	1.00	0.00	-0.10	-0.40
9 | u | 1.22	5.67	-0.56	1.00	0.00	-0.10	-0.33
10 | $x_4 * x_4 * sin(x_2)$ | 1.22	5.63	-0.56	1.00	0.00	-0.10	-0.33

(a) Best models for $\dot{\mathbf{x}}_3$. trajectories.



	$sin(x_2)$	$\dot{x}_4$	$x_3 * cos(x_2)$	$cos(x_2) * u$	$cos(x_2) * cos(x_2) * \dot{x}_4$	$x_4 * x_4 * sin(x_2) * cos(x_2)$
0 | $x_3 * cos(x_2)$ | 3.17	32.20	1.00	-19.96	1.86	-0.31	0.33
1 | $x_4 * x_4 * sin(x_2) * cos(x_2)$ | 3.15	32.10	1.00	-19.27	1.85	-0.31	0.34
2 | $sin(x_2)$ | 3.11	32.76	1.00	-18.88	1.81	-0.33	0.32
3 | $cos(x_2) * u$ | 3.14	32.22	1.00	-19.48	1.87	-0.31	0.33
4 | $cos(x_2) * cos(x_2) * \dot{x}_4$ | 3.47	31.02	1.00	-17.01	1.62	-0.43	0.30
5 | $\dot{x}_4$ | 3.07	31.93	1.00	-18.72	1.79	-0.34	0.32

(b) Best models for $\dot{\mathbf{x}}_4$.

Fig. 41: The best identified models for both accelerations.

The accurate models were identified many times using different guesses θ_i. The identified angular acceleration model has low error, but it's missing the term x_4, which represents the pendulum joint friction. I set the joint friction coefficient at a very low value, so its effects were negligible and apparently got lost in the noise. The model for cart acceleration is missing the $x_4 \cos(x_2)$ term, whose parameter is very low 0.02 in the exact model. This term represents the change in cart acceleration caused by pendulum joint friction.

9.7 Parameter tuning

The models identified in Figure 41 all have the same structure, but there's some variance in their parametrization. It appears that the parameters change depending on the guess function θ_i used to generate the model. Now that we used SINDy to identify which functions are active in the dynamics, we can use other, more common non-sparse regression methods to fine-tune the function parameters. The regression task is redefined so that the function library $\mathbf{\Theta}$ only contains the functions we've already identified. The required solution $\boldsymbol{\xi}$ no longer needs to be sparse, and the solution vector will also be shorter. Because the identified models are both rational, we still have to use the SINDy-PI regression formulation - extracting a column θ_i from the function library $\mathbf{\Theta}$, setting it as the target variable and calculating the solution $\boldsymbol{\xi}_i$.
The solution $\boldsymbol{\xi}$ will slightly change with different measurement data used to construct the function library $\mathbf{\Theta}$ and with the regularization hyperparameter λ. I'll generate 25000 different models for each of the acceleration models and then plot histograms of the respective function parameters to see how consistent they are. If the regression was poorly conditioned, then a small change in training data would result in a large change in the parameters. I'll generate the different models by changing (resampling) the training data sets.
First, I'll generate the function library $\mathbf{\Theta} \in \mathbb{R}^{N \times m}$, but only with the candidate functions that were already picked. Then I'll re-sample the library by picking N **random** samples (rows) from the library and creating a new, resampled version of $\mathbf{\Theta}$. Then, to be consistent with the SINDy-PI formulation, I pick a random column θ_i from $\mathbf{\Theta}$, extract it, set it as a target variable and find the solution $\boldsymbol{\xi}_i$ using Ridge regression. The random resampling and regression step is then repeated 25000 times, saving the results after every iteration.
This method of randomly drawing samples from one data set to generate a large number of models is called **bootstrapping** [22]. It is commonly used to generate confidence intervals for the parameters of linear models.
All the model parameters are normalized so the acceleration function parameter is always 1. This doesn't change the model, because the explicit model is rational. The histograms for model parameters of both acceleration models are shown in Figures 42 and 43. Each plot corresponds to a single parameter, the colors of the histograms specify which guess function θ_i was used to generate the solution.

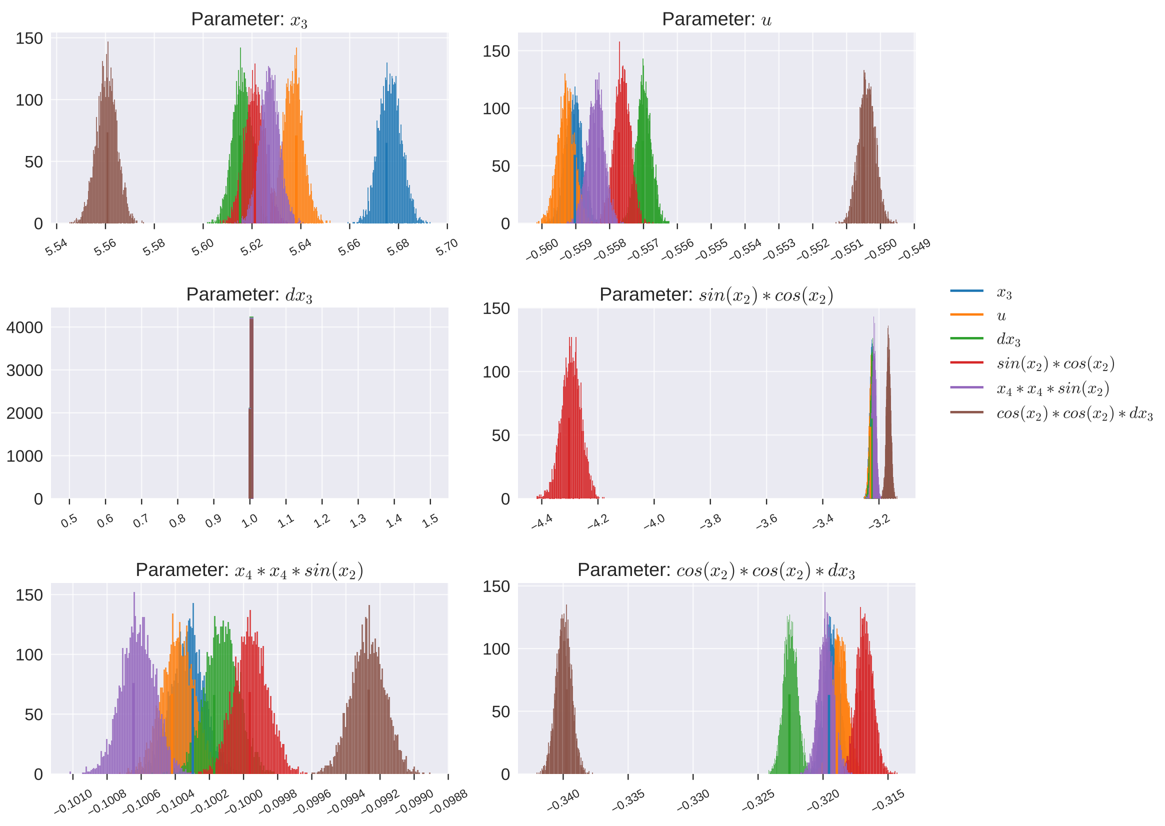


Fig. 42: Bootstrapping results for cart acceleration model parameters. The color of the distribution depends on the guess function used to generate the model.

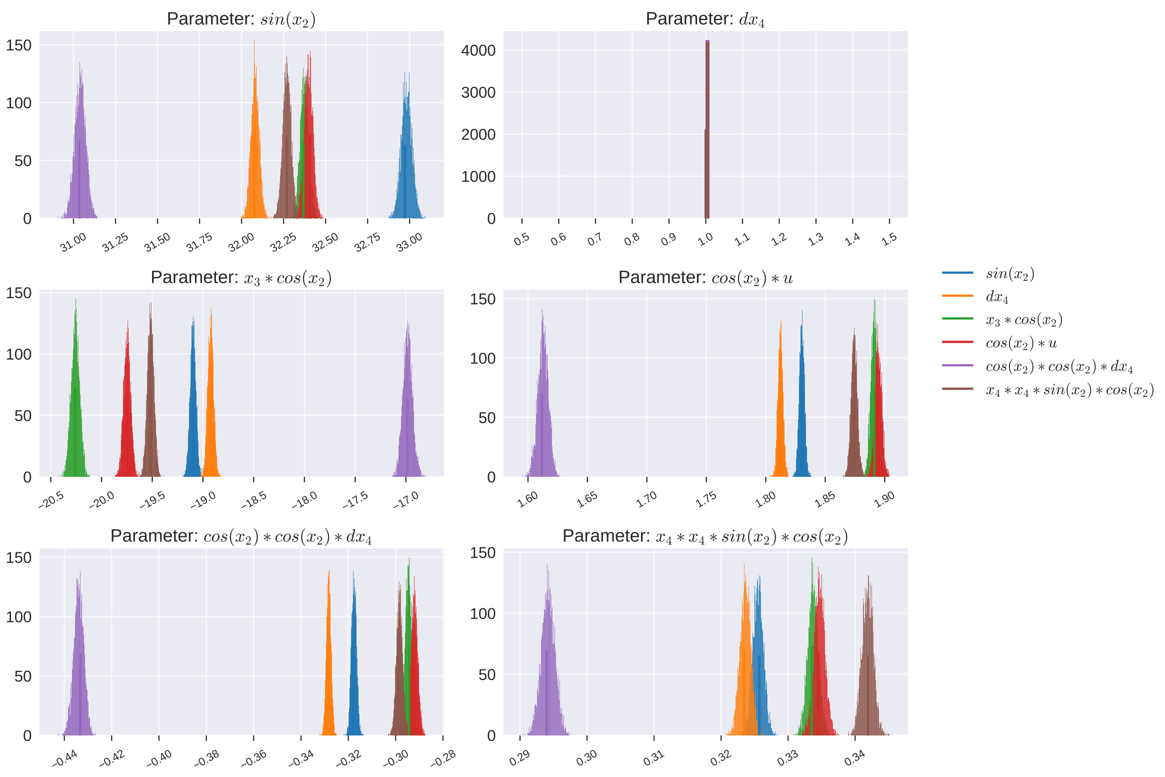


Fig. 43: Bootstrapping results for pendulum angular acceleration model parameters.

From Figures 42 and 43, it's apparent that the parameters sometimes change significantly based on the used guess function θ_i. The parameter distributions are very different when the functions $\cos^2(x_2)\dot{x}_3$ or $\cos^2(x_2)\dot{x}_4$ are used as guess functions for their respective models. These functions have two things in common - both contain $\cos^2(x_2)$ and both contain the respective acceleration $\dot{x}_i$. It's not clear to me whether the parameter distribution shift is caused by one of these things or whether it's a coincidence.

The specific model parameters will be chosen from the histograms in Figures 44 and 45 which don't differentiate the models based on the guess functions.

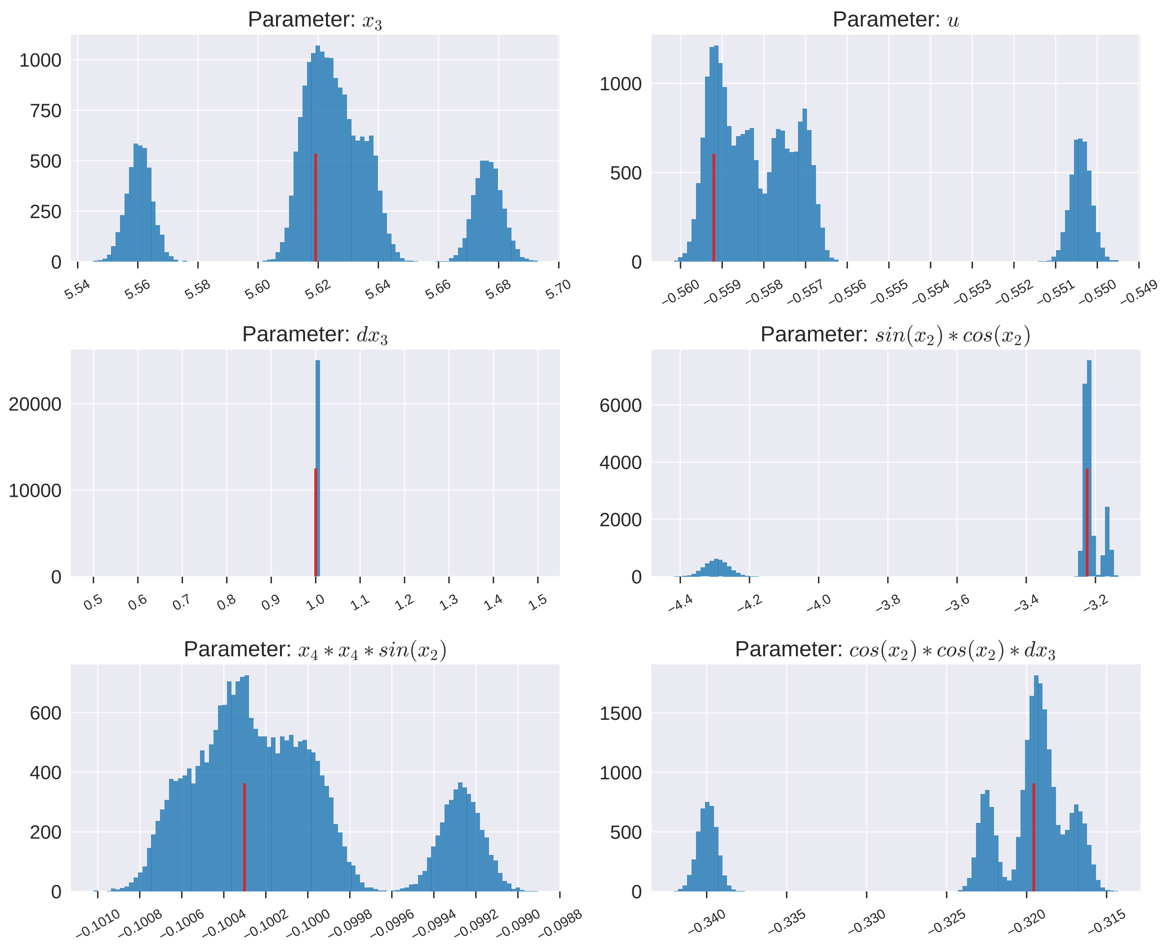


Fig. 44: Combined bootstrapping results for cart acceleration model parameters.

Now that we have the parameter distributions, we can generate a new model by picking the parameters. How to pick the parameters is not clear. If the distributions were unimodal, then I'd definitely pick the parameters as the medians of the respective distributions. In this highly polymodal case, the mode seems as the optimal statistic. The modes are visualized in the Figures 44 and 45 by the red vertical lines.

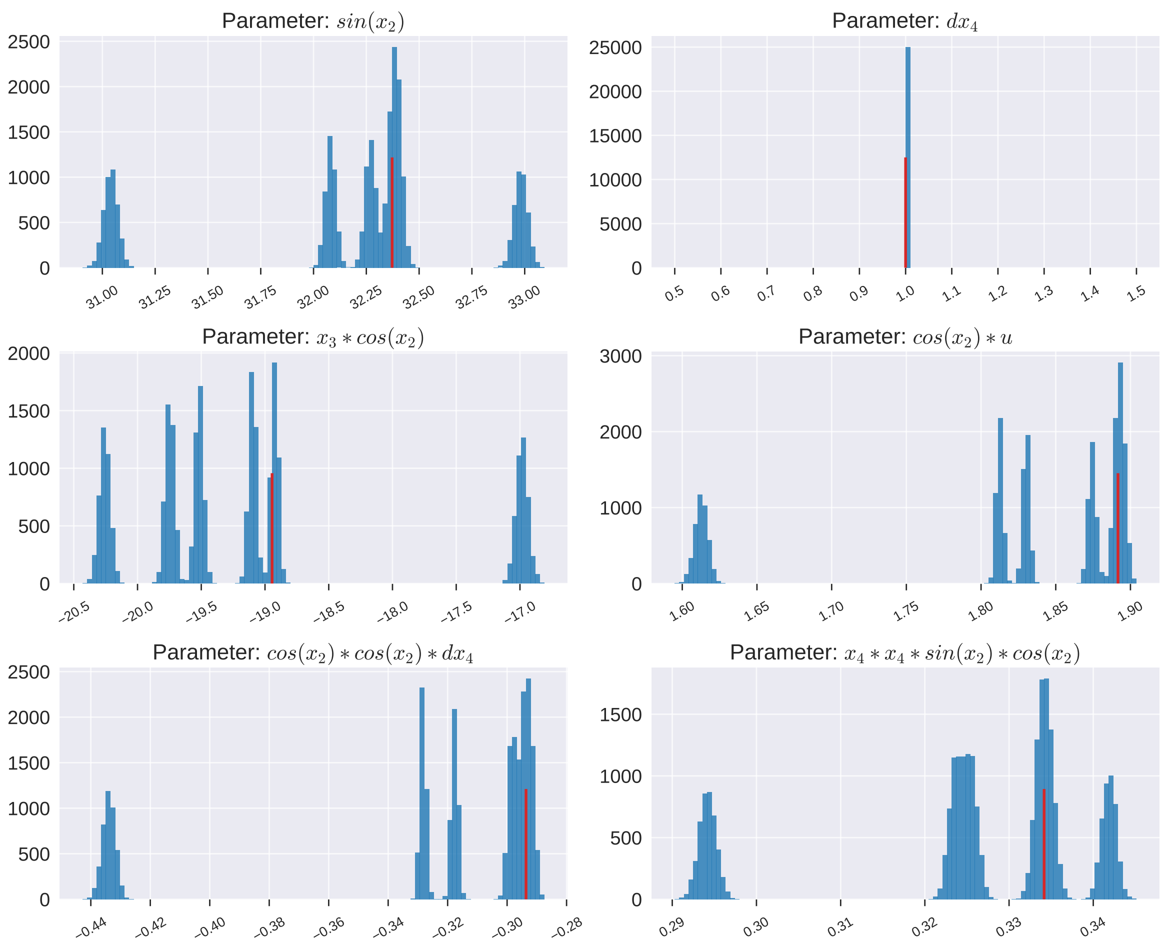


Fig. 45: Combined bootstrapping results for pendulum angular acceleration model parameters.

9.8 Validation

The reference analytical model and both identified models are shown in the Table 4.

Table 4: Comparison of the reference and the identified models.

	Cart acceleration model $\dot{x}_1$ $[\frac{\mathrm{m}}{s^2}]$
Reference	$\frac{-0.66407u+6.64064x_3-0.11953x_4^2\sin(x_2)-0.02169x_4\cos(x_2)-1.91599\sin(2.0x_2)}{0.19531\cos(2.0x_2)-1.0}$
SINDy	$\frac{-0.55672u+5.61287x_3-0.10007x_4^2\sin(x_2)-1.61162\sin(2.0x_2)}{0.32312\cos^2(x_2)-1.0}$
Bootstrapped	$\frac{-0.55871u+5.62374x_3-0.10038x_4^2\sin(x_2)-1.6134\sin(2.0x_2)}{0.31937\cos^2(x_2)-1.0}$
	Pendulum angular acceleration model $\dot{x}_2$ $[\frac{\mathrm{rad}}{s^2}]$
Reference	$\frac{1.81554u\cos(x_2)-18.15533x_3\cos(x_2)+0.1634x_4^2\sin(2.0x_2)+0.18156x_4+32.05886\sin(x_2)}{0.3268\cos^2(x_2)-1.0}$
SINDy	$\frac{2.16696u\cos(x_2)-22.63092x_3\cos(x_2)+0.19338x_4^2\sin(2.0x_2)+38.39076\sin(x_2)}{0.19753\cos(2.0x_2)-1.0}$
Bootstrapped	$\frac{2.21986u\cos(x_2)-22.3136x_3\cos(x_2)+0.19665x_4^2\sin(2.0x_2)+37.93954\sin(x_2)}{0.17497\cos(2.0x_2)-1.0}$

The discovered models will now be validated using a validation data set generated by a separate simulation. First, let's test the model's derivative estimation accuracy.

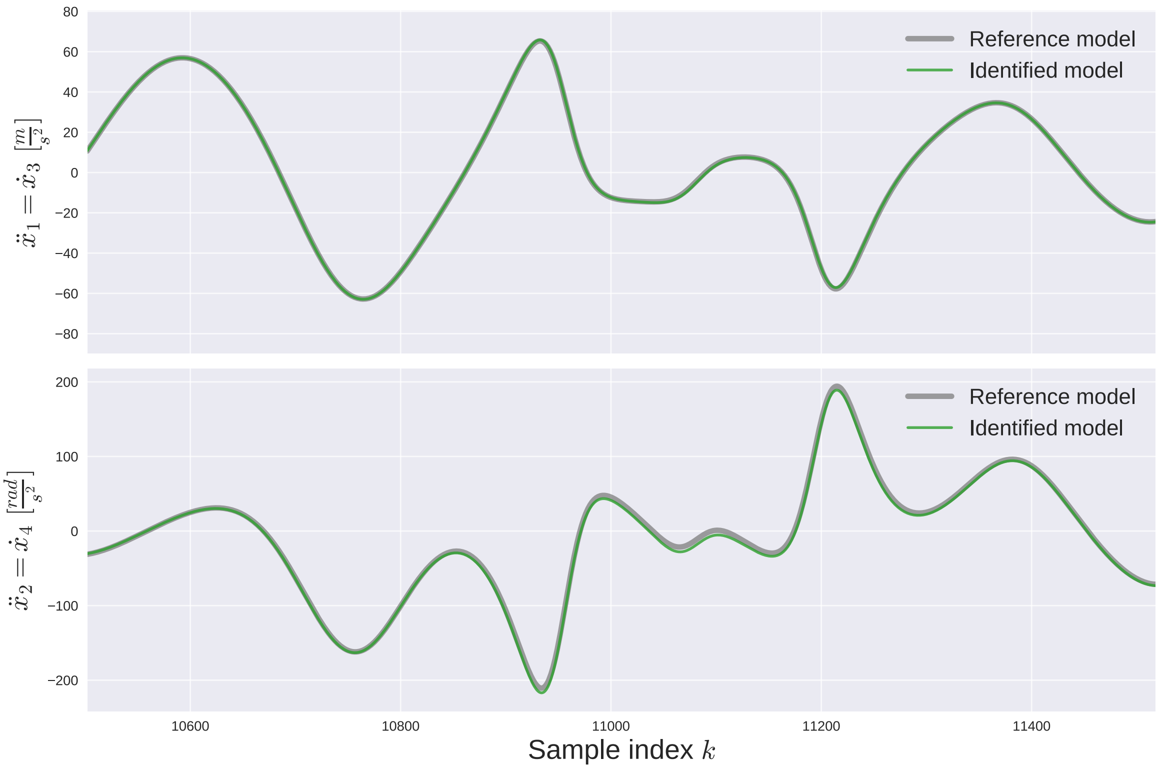


Fig. 46: Validation of the identified cart-pendulum SINDy model's state derivative estimation accuracy. The estimations are very close to the test set values.

The RMSE metrics values for all acceleration models:

Table 5: Validation RMSE for each identified acceleration model $\dot{x}_i$.

Model	SINDy model	Bootstrapped SINDy model
$\mathrm{RMSE}(\dot{x}_1 \mid \boldsymbol{\xi})$	1.46E0	2.03E0
$\mathrm{RMSE}(\dot{x}_2 \mid \boldsymbol{\xi})$	2.57E0	2.61E0

All validation RMSE metrics are lower than the training RMSEs. The explanation for this is the same as in the example with the Lorenz system - the validation data is clean, but the training data isn't. A big portion of the training error was caused by inaccuracies in the acceleration signal's estimation.

The models will be further evaluated by simulating two models and comparing the results. The reference model is the one that was used to generate the training and validation data. Both models in the simulations will start from the same initial state and will receive the same input signal. Despite the derivative estimation errors being very small for both identified models, small estimation inaccuracies accumulate (exponentially) and the trajectories decouple. This is a direct consequence of the chaotic nature of the pendulum-cart system.

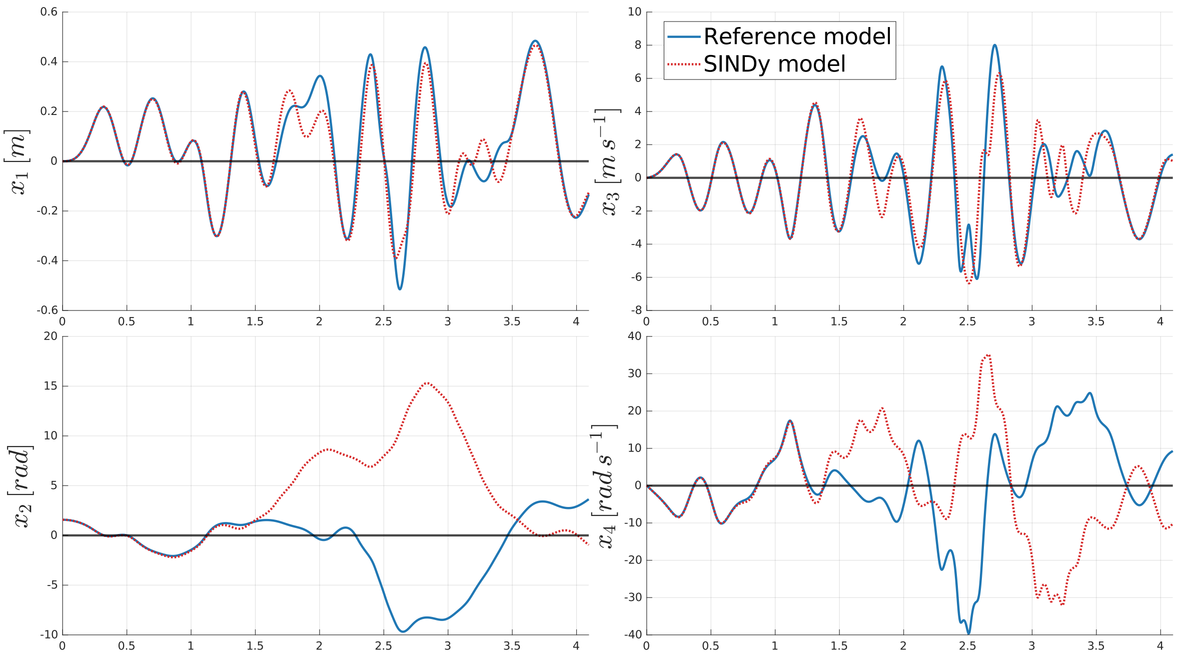


Fig. 47: Parallel simulation of the reference model and the SINDy model. Full animation at: https://git.io/JBPXD

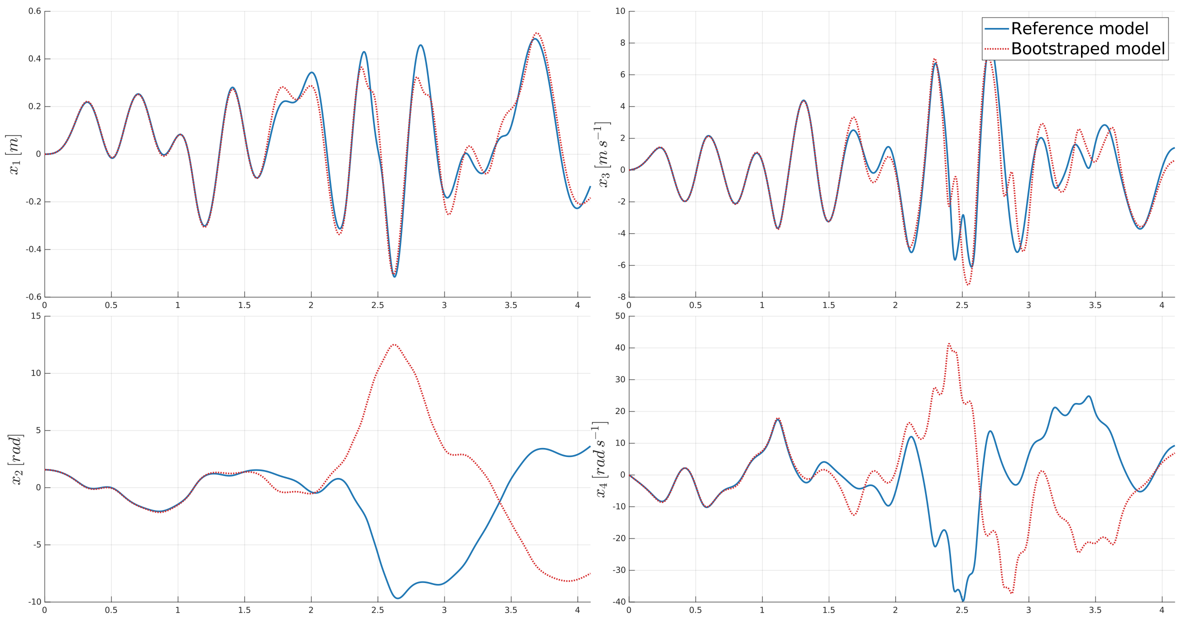


Fig. 48: Parallel simulation of the reference model and the bootstrapped SINDy model. The results are indistinguishable from the previous simulation. Full animation at: https://git.io/JBPXA

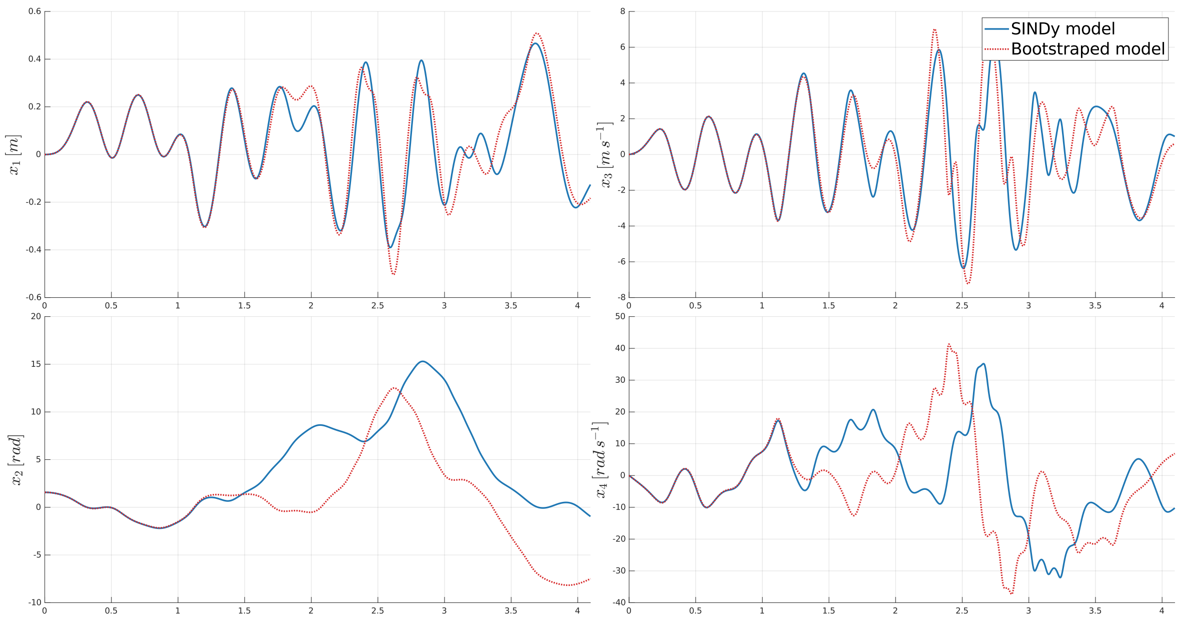


Fig. 49: Parallel simulation of the best SINDy model and the bootstrapped SINDy model. Despite both models being identical in their structure and very close parameter-wise, the trajectories still decouple after a few seconds. Full animation at: https://git.io/JBP1L

10 Model identification of a real pendulum-cart system

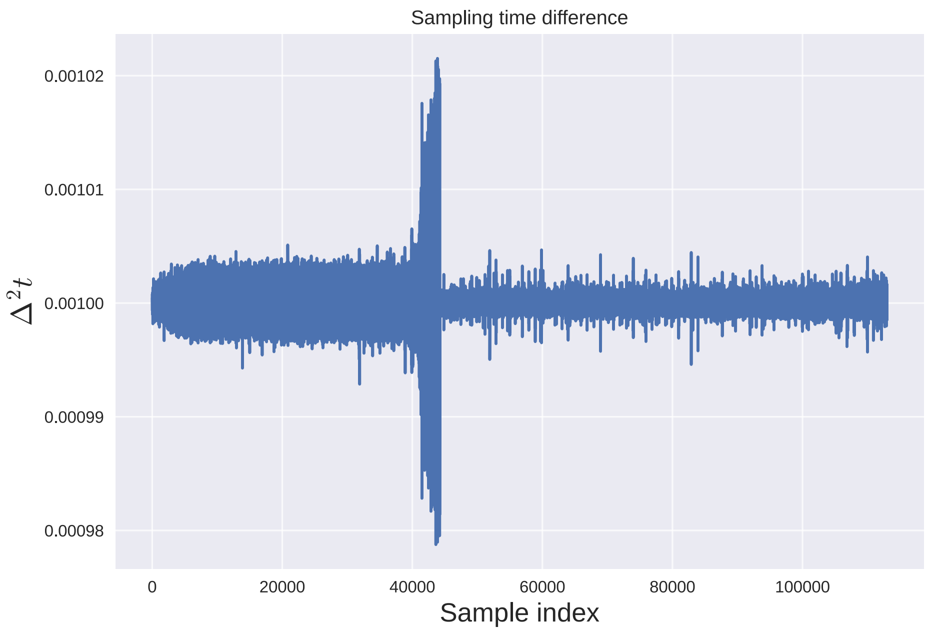
10.1 Real system

In this section, the method will be used for identifying a model from measurements of a real cart-pendulum system. The physical system was provided by REX Controls [23]. The system has three pendulums mounted on a cart; for my purposes, the pendulums were tied up, so there was only one free angular degree of freedom. The physical system is also provided with control HW and SW. By default, the SW contains control algorithms for a triple pendulum swing-up maneuver. The algorithms are implemented in REXYGEN Studio, the company's own development environment. The swing-up maneuver algorithm works by sending a pre-computed acceleration trajectory to the linear drive's control system. I also needed to send a pre-computed trajectory to the system, except in my case, the trajectory is a random band-limited process. Instead of lengthy implementation of a separatate functionality for sending arbitrary trajectories, I forked the existing SW and changed the swing-up control block to contain my own trajectory instead of the swing-up one. That way, I could also use the existing front-end when running the experiments. The experiment could be started by simply clicking on the "Start swingup" button in the user interface.

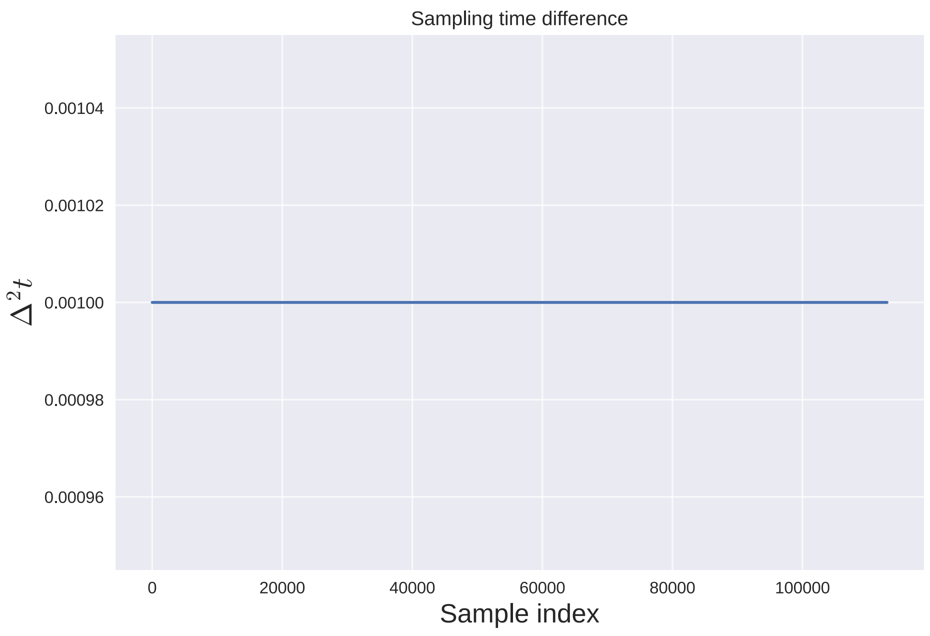
10.2 Data collection and processing

When creating the trajectories to be used for the experiments, I had to make sure the physical constraints are met. Those included, most notably, the limited length of the track $(\pm 0.5m)$ and linear acceleration. I already developed a method to compute trajectories that had limited deviance from the origin in the earlier sections. I only had to make sure that the resulting acceleration signal never exceed $4\mathrm{G}$, or around $40\frac{\mathrm{m}}{\mathrm{s}^2}$. The maximum acceleration can be implicitly controlled by the band-width of the generated position trajectory signal. For safety purposes, my pre-computed trajectory never exceeded $2.5\mathrm{G}$.
The system's control system has a sampling frequency of about $1\mathrm{kHz}$, so the state measurements were of pretty good quality. Any noise present in the state measurements could relatively easily be filtered out. One small problem was that the measurements weren't taken *exactly* every $0.001\mathrm{s}$, the sampling periods sometimes varied by $2\mathrm{ms}$. For FFT, the sampling period should be constant, so I cleaned up the data by simply re-sampling it using linear interpolation. The time difference between samples before and after the resampling is shown in Figures 50

The measured signals were the cart position and velocity x_1, x_3, pendulum angle and angular velocity x_2, x_4 and the input u. While the state measurements were clean, the input u was extremely noisy and had a shifted 0. At the start of the experiment, when the reference trajectory wasn't even sent in and the system wasn't moving at all, the input u was non-zero. To make matters worse, at the end of the experiment, where the cart was no longer moving, the input u was still non-zero and had a different value than at the start of the experiment. Therefore, I couldn't remove the offset by subtracting it.



(a) Before resampling



(b) After resampling

Fig. 50: The time difference between samples before and after resampling.

I at least subtracted the mean value, so the input's mean value was 0.
All measured signals, including the input u, were filtered using a spectral filter with cutoff frequencies defined in Figure 51. The x_5 signal is the input u.

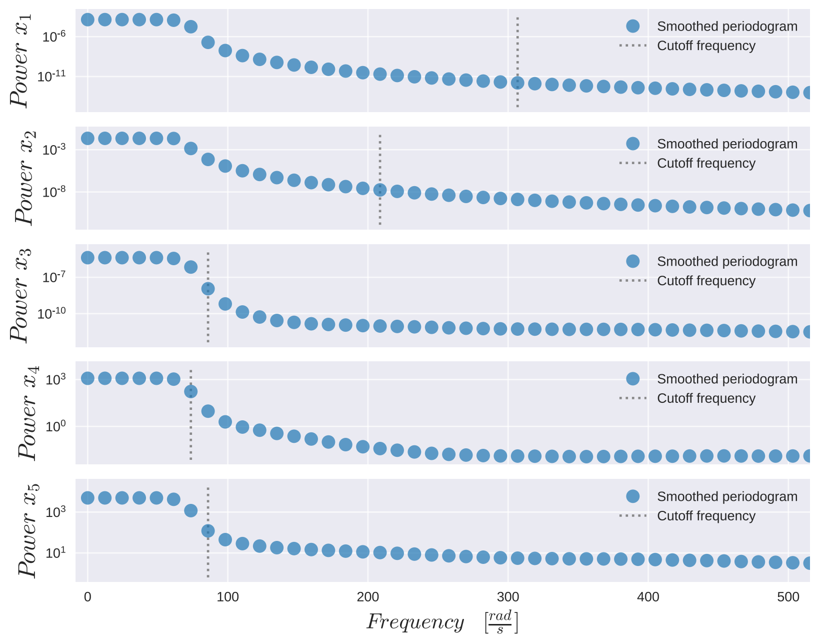
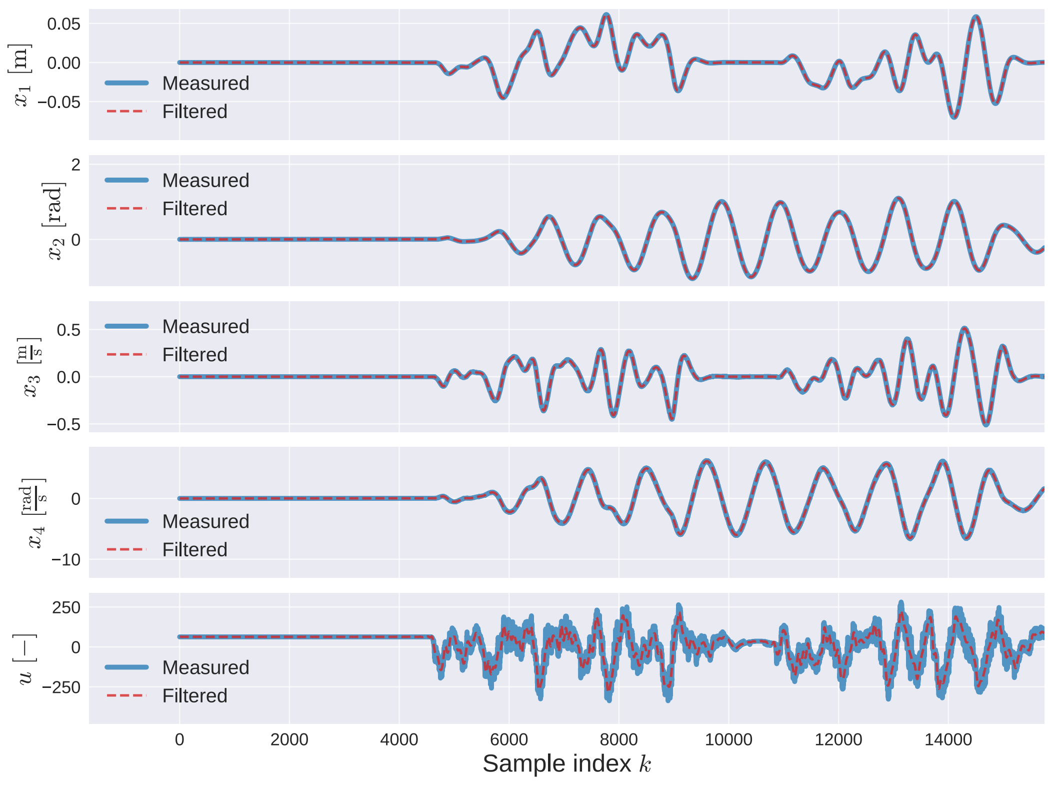
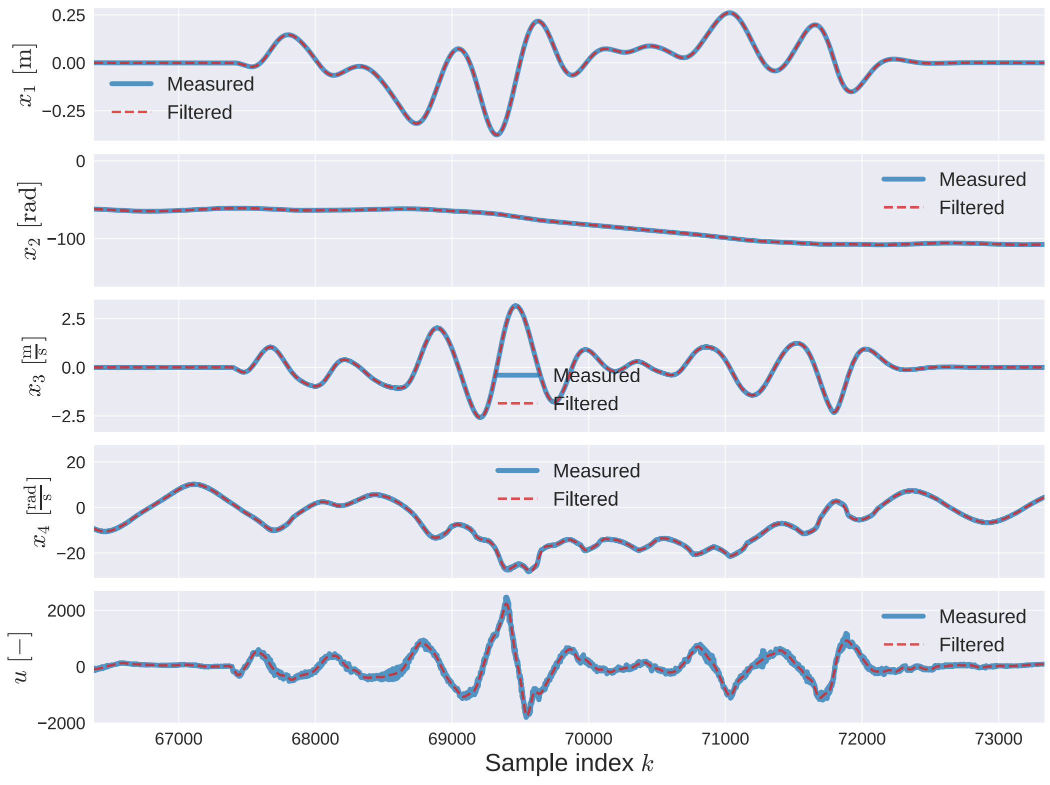


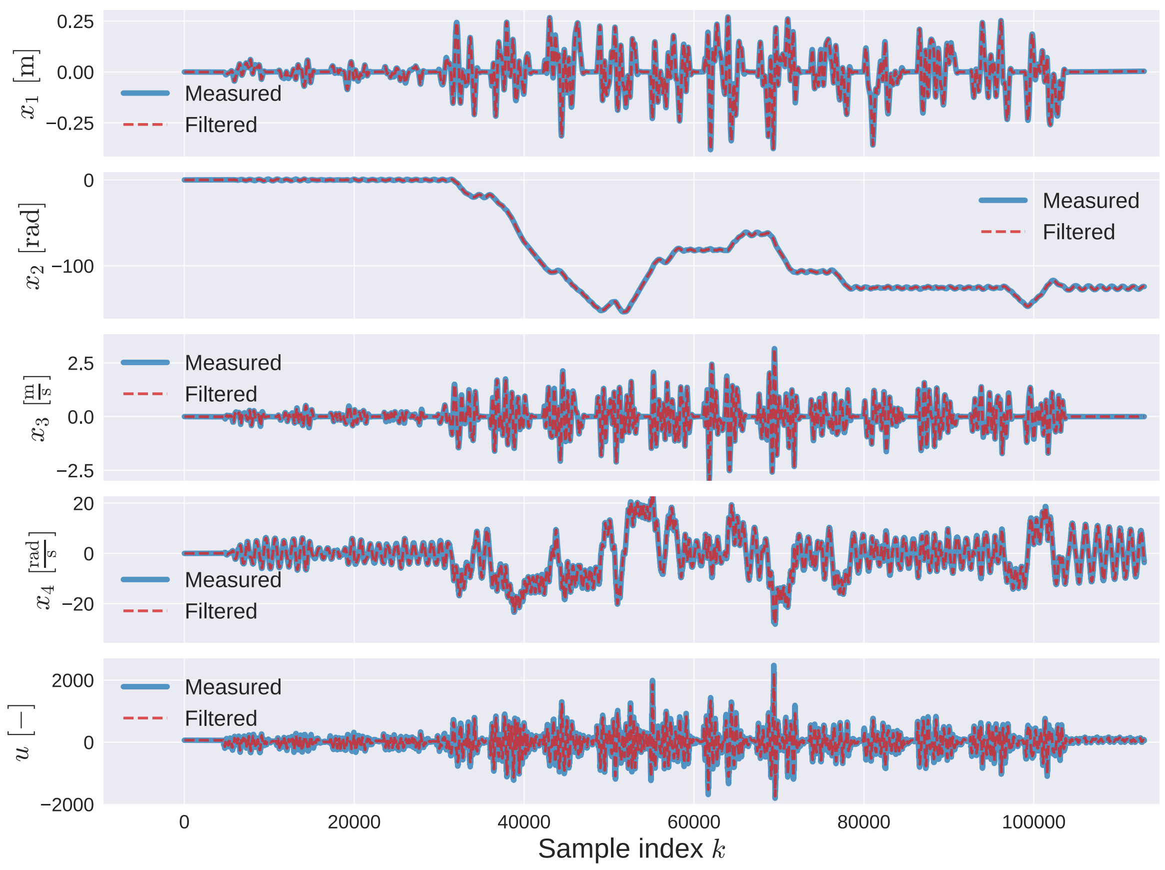
Fig. 51: Filter cutoff frequencies. The filters for the velocity signals x_3 and x_4 have much lower cutoff frequencies, since the signals will be used for numerical differentiation to estimate accelerations.



(a) The first few seconds of the experiment. Note the shift in the input u when



(b) Part of the measurement signals.



(c) The full measurements

Fig. 51: The state and input measurements, before and after filtering. The measurement data is also visualized at: https://git.io/JBbux

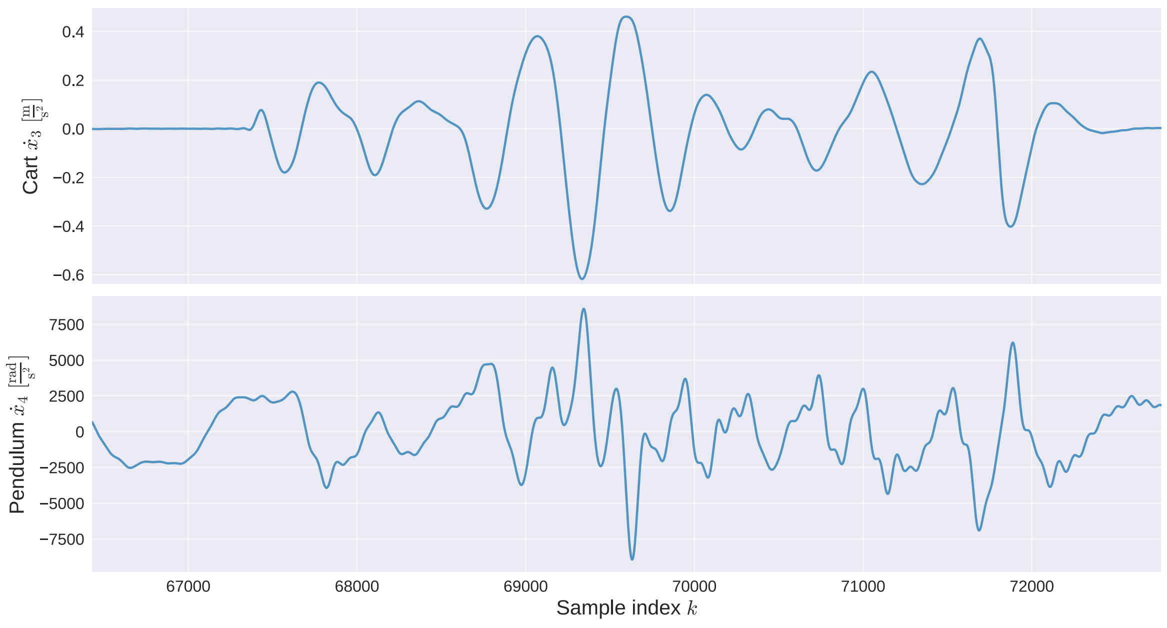


Fig. 52: Accelerations $\dot{x}_3$ and $\dot{x}_4$ estimated from velocity measurements.

10.3 Identification

The function library $\mathbf{\Theta}$ was then constructed the state measurements $\mathbf{X}$, state derivative estimates $\dot{\mathbf{X}}$ and input measurements $\mathbf{U}$. The candidate function library contained the following functions:

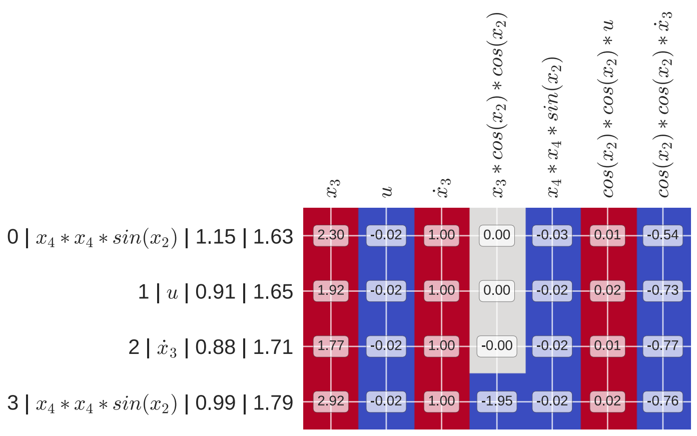
```
Theta.columns = [
        '1', 'x_3', 'x_4', 'sin(x_2)', 'cos(x_2)', 'u', 'dx_3', 'dx_4',
        'x_3*x_3', 'x_3*sin(x_2)', 'x_3*cos(x_2)', 'x_4*x_4', 'x_4*sin(x_2)',
        'x_4*cos(x_2)', 'sin(x_2)*cos(x_2)', 'sin(x_2)*u', 'sin(x_2)*dx_3',
        'sin(x_2)*dx_4', 'cos(x_2)*cos(x_2)', 'cos(x_2)*u', 'cos(x_2)*dx_3',
        'cos(x_2)*dx_4', 'x_3*x_3*sin(x_2)', 'x_3*x_3*cos(x_2)',
        'x_3*sin(x_2)*cos(x_2)', 'x_3*cos(x_2)*cos(x_2)', 'x_4*x_4*sin(x_2)',
        'x_4*x_4*cos(x_2)', 'x_4*sin(x_2)*cos(x_2)', 'x_4*cos(x_2)*cos(x_2)',
        'sin(x_2)*cos(x_2)*u', 'sin(x_2)*cos(x_2)*dx_3',
        'sin(x_2)*cos(x_2)*dx_4', 'cos(x_2)*cos(x_2)*u',
        'cos(x_2)*cos(x_2)*dx_3', 'cos(x_2)*cos(x_2)*dx_4',
        'x_4*x_4*sin(x_2)*cos(x_2)']
```

By varying the sparsity hyperparameter λ_R and by choosing different guess functions θ_i, a total of 960 models was generated for each of the accelerations. Many of the models were exactly identical (in active functions and parameters), so only the unique ones were kept. Non-sparse models and models with a very high training error were discarded. The best discovered models for the cart's linear acceleration $\dot{x}_3$ and the pendulum's angular acceleration $\dot{x}_4$ are shown in Figures 53.

10.4 Bootstrapping

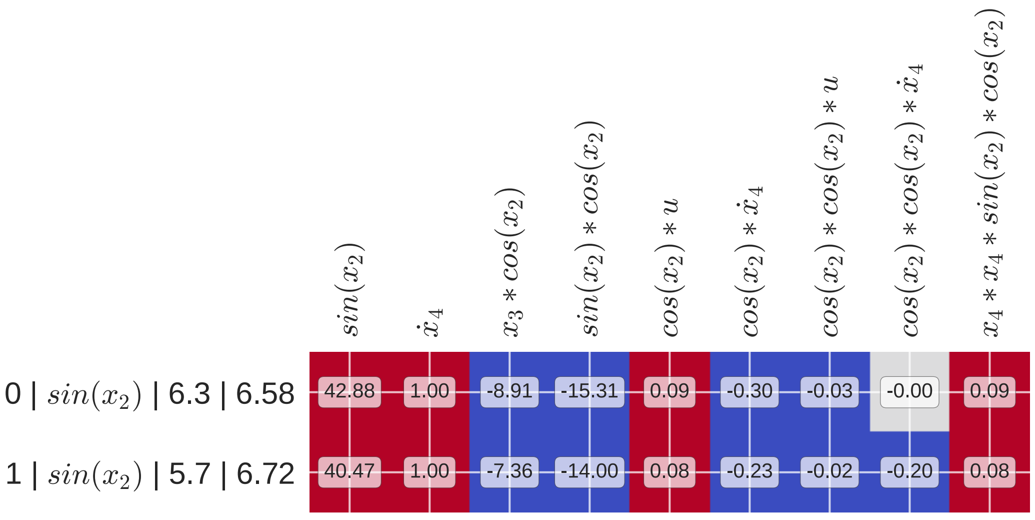
Using only the active terms, I constructed a new function library Θ and trained 10000 models for each acceleration model using the bootstrapping method described earlier in (9.7). The guess function θ_i is also chosen randomly for each model. That way, the parameter estimates between regression results using different guess functions can be compared.

From the Figures (54), it's apparent that some parameters for $\dot{x}_3$ change drastically with different candidate function guesses. This is indicative of the terms not being picked accurately by the sparse regression.



	x_3	u	$\dot{x}_3$	$x_3 * cos(x_2) * cos(x_2)$	$x_4 * x_4 * sin(x_2)$	$cos(x_2) * cos(x_2) * u$	$cos(x_2) * cos(x_2) * \dot{x}_3$
0 | $x_4 * x_4 * sin(x_2)$ | 1.15 | 1.63	2.30	-0.02	1.00	0.00	-0.03	0.01	-0.54
1 | u | 0.91 | 1.65	1.92	-0.02	1.00	0.00	-0.02	0.02	-0.73
2 | $\dot{x}_3$ | 0.88 | 1.71	1.77	-0.02	1.00	-0.00	-0.02	0.01	-0.77
3 | $x_4 * x_4 * sin(x_2)$ | 0.99 | 1.79	2.92	-0.02	1.00	-1.95	-0.02	0.02	-0.76

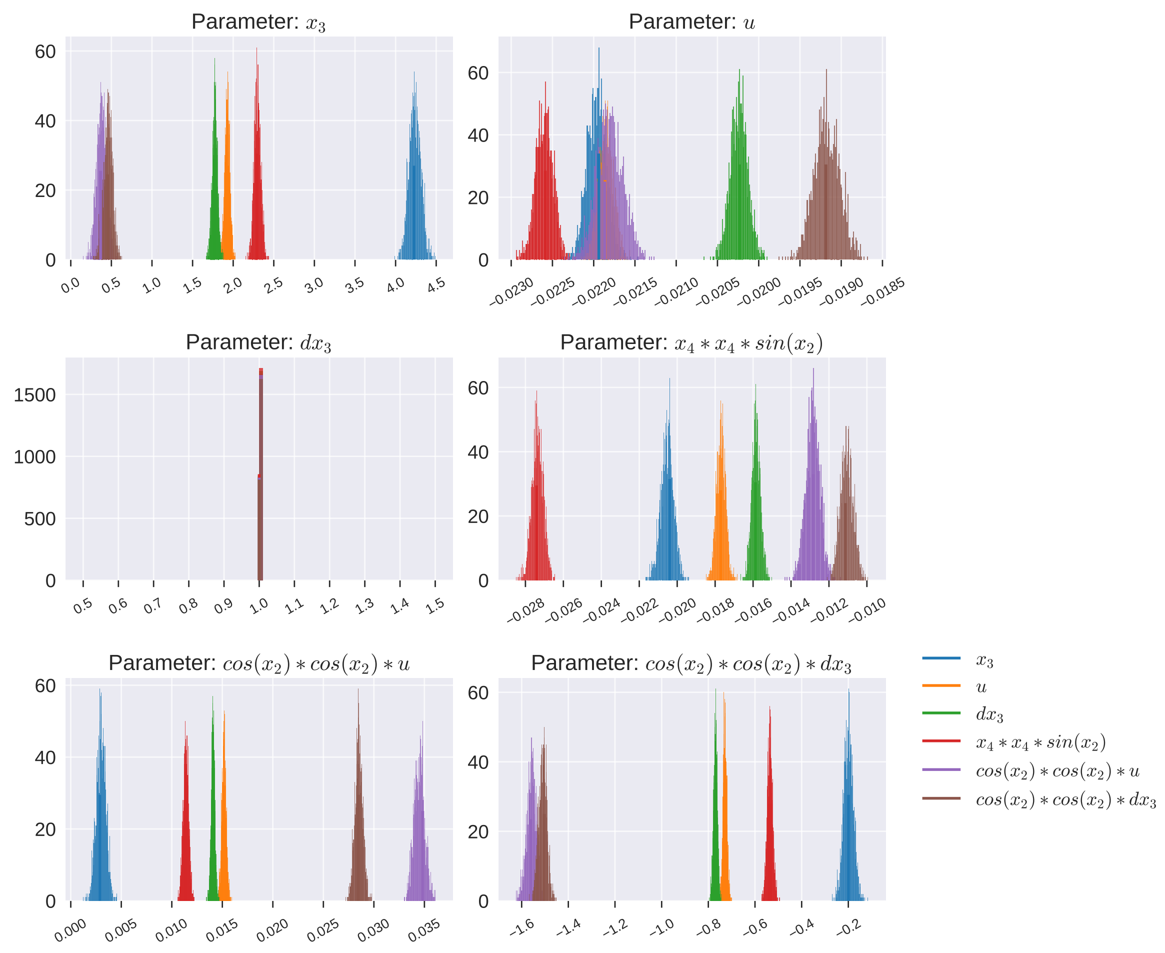
(a) The best discovered models for the linear cart acceleration $\dot{x}_3$.



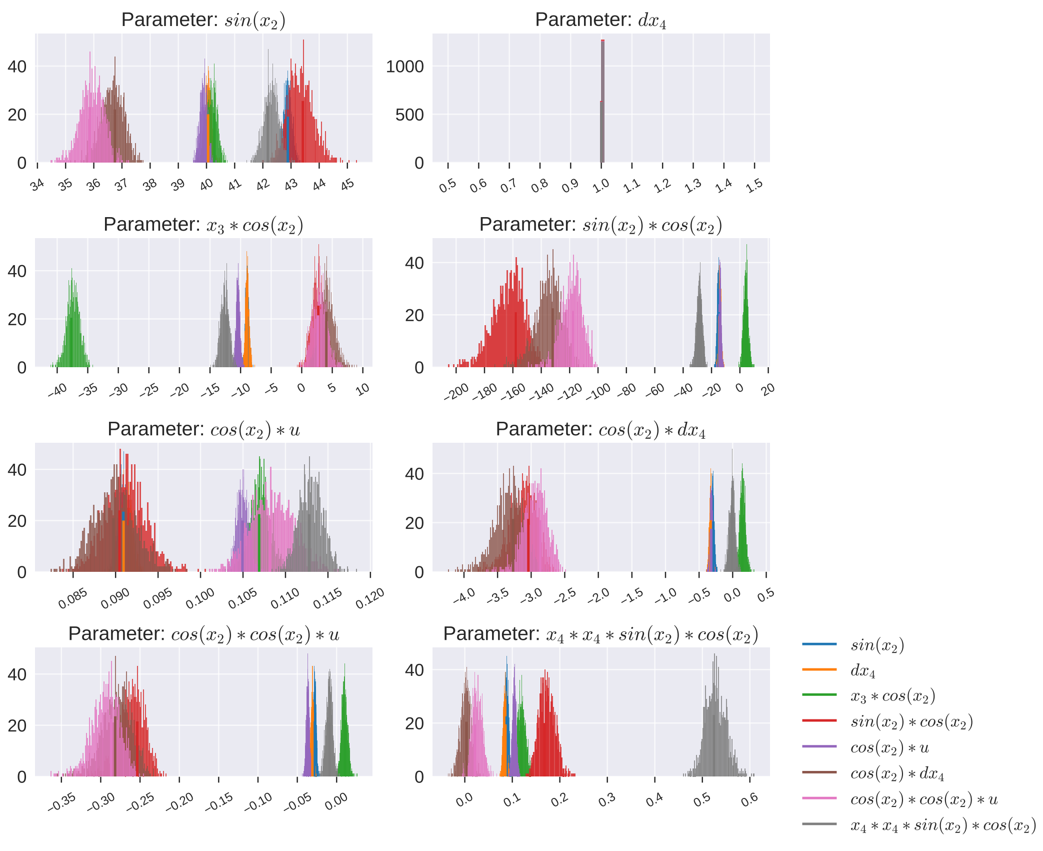
	$sin(x_2)$	$\dot{x}_4$	$x_3 * cos(x_2)$	$sin(x_2) * cos(x_2)$	$cos(x_2) * u$	$cos(x_2) * \dot{x}_4$	$cos(x_2) * cos(x_2) * u$	$cos(x_2) * cos(x_2) * \dot{x}_4$	$x_4 * x_4 * sin(x_2) * cos(x_2)$
0 | $sin(x_2)$ | 6.3 | 6.58	42.88	1.00	-8.91	-15.31	0.09	-0.30	-0.03	-0.00	0.09
1 | $sin(x_2)$ | 5.7 | 6.72	40.47	1.00	-7.36	-14.00	0.08	-0.23	-0.02	-0.20	0.08

(b) The best discovered models for the pendulum's angular acceleration $\dot{x}_4$.

Fig. 53: Discovered models for the real pendulum-cart system. The y-axis labels contain the model's index, the quess function used to generate the model, the training error and the validation error.

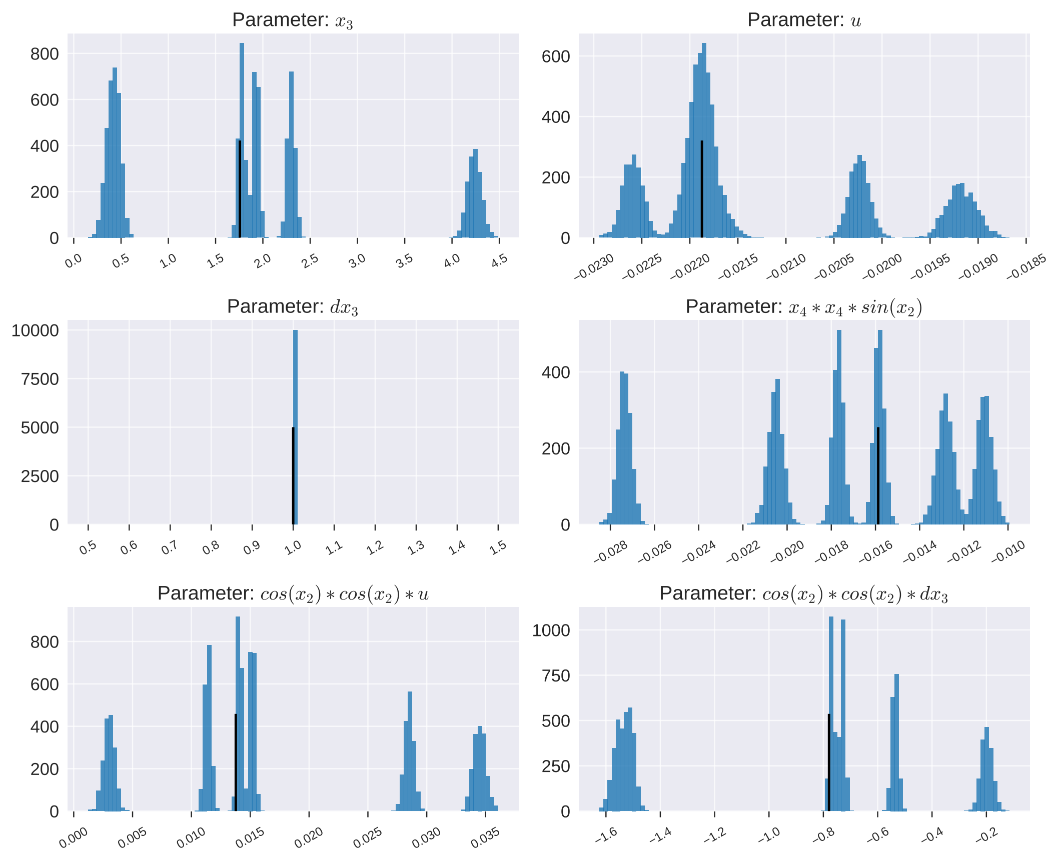


(a) Parameters for $\dot{x}_3$ for different candidate function guesses.

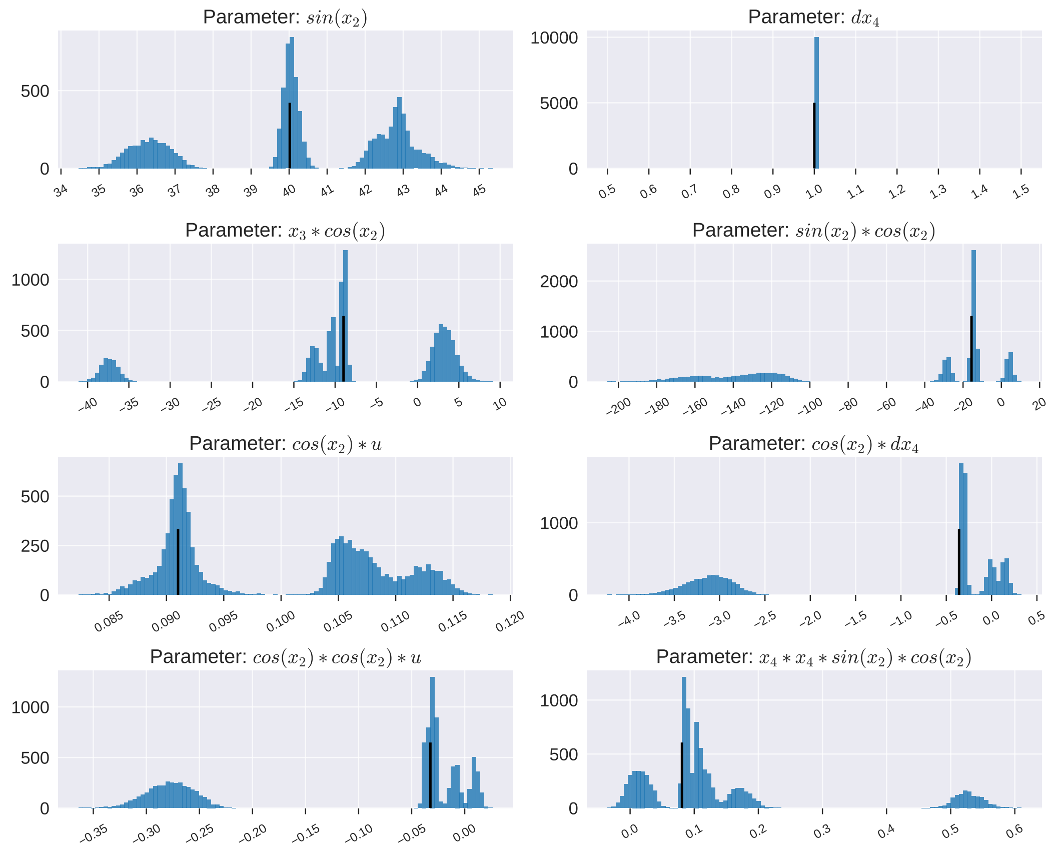


(b) Parameters for $\dot{x}_4$ for different candidate function guesses.

Fig. 54: Parameter distributions for both models for different candidate function guesses.



(a) Combined distributions for $\dot{x}_3$.

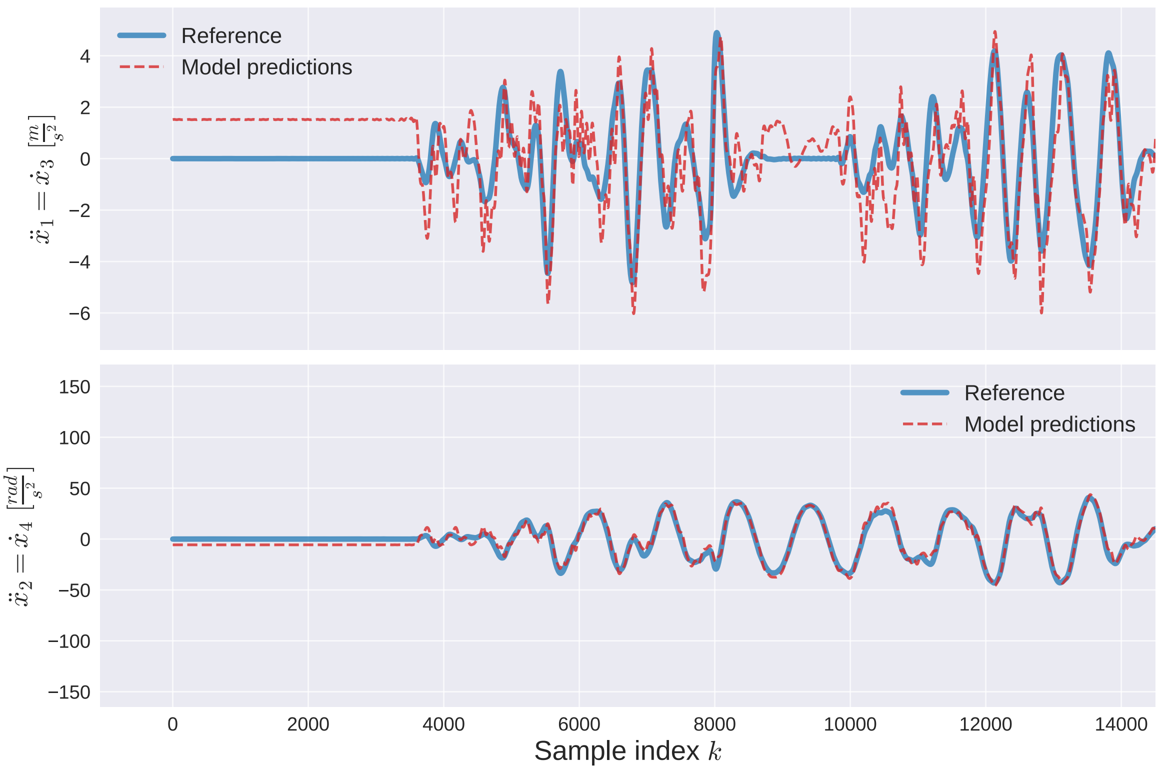


(b) Combined distributions for $\dot{x}_3$.

Fig. 55: Combined parameter distributions for both models. The black vertical lines show the modes of the distributions.

10.5 Validation

From the SINDy models in the previous Figure 53, I chose the models with the lowest validation RMSE. The validation RMSE is the last value on the y-axis labels. The plots with derivative estimates are in Figure 55.



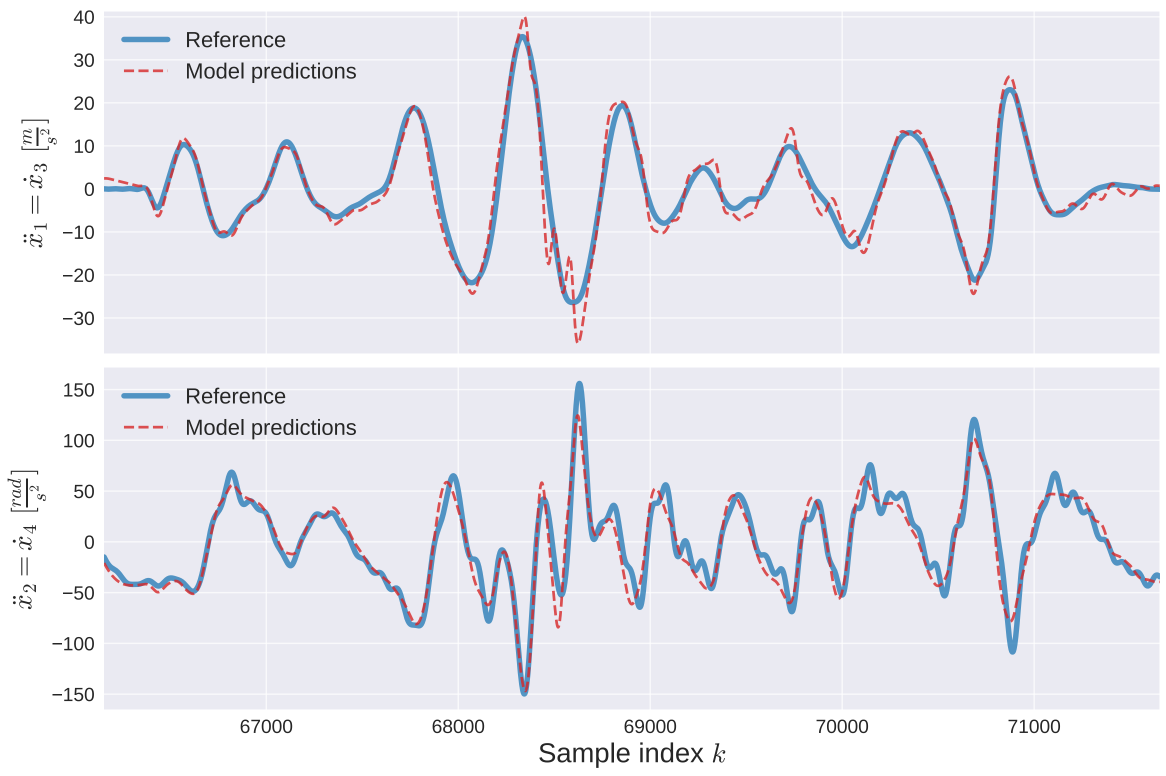
(a) Derivative estimates for the start of the experiment

The accuracy of the derivative estimate is highly dependent on the state and input. When the state is at rest, the estimates are very bad. This is caused by the bad quality of input measurements, which were non-zero at rest. Another problem is that the input measurements weren't measuring the action variable directly, instead they were the inputs (references) into the cart's control system. Ideally, the input measurements would be the outputs of the control system. Identifying the controlled system's dynamics accurately using only set-point (inputs to the control system) measurements might be possible by including additional candidate functions θ. The candidate function library $\boldsymbol{\Theta}$ was designed assuming that the system is a pendulum-cart system with force u as an input, but that assumption doesn't fully hold in this case.

The results of the model's simulation are in the Figure 56.

Qualitatively, the model simulations look realistic. The real system had 3 pendulums, which were folded and tied up to create a single pendulum system. Because of this, the pendulum has a relatively high moment of inertia, as is apparent from the simulation results.

The model parametrized by the parameter distribution modes from the bootstrapping suffered from the same errors. Its validation RMSE was comparable to the SINDy model's RMSE.



(b) Derivative estimates later in the experiment

Fig. 55: The discovered model's derivative estimates compared to the reference values estimated from the training data using numerical differentiation. Due to heavy noise present in the u measurements and its offset from 0, the derivative estimates are also noisy and offset.

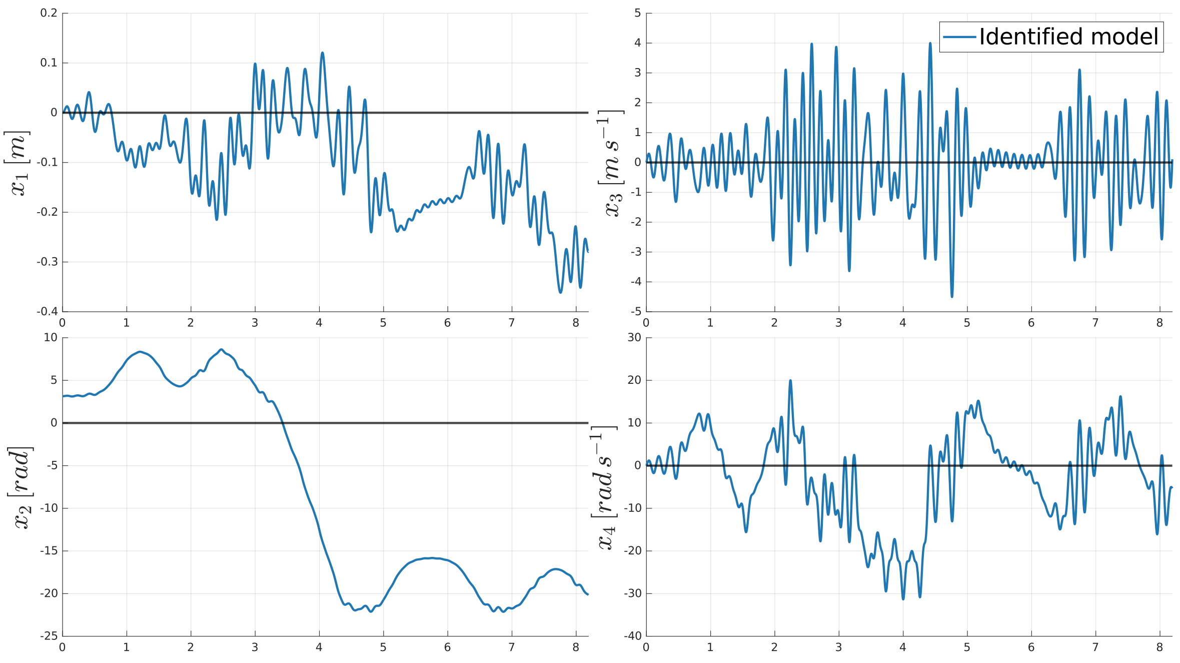


Fig. 56: Simulations of the model identified from real data. The animation is at: https://git.io/JBbUT

11 Conclusion

In this thesis, the SINDy-PI method was used to discover rational, nonlinear and sparse models from state and input measurement data.

Before applying the method itself, I introduced a small change to the regression method. The previous method, called sequentially thresholded least squares (STLS), finds sparse solutions to the given regression problem by first computing the least squares solution and then setting parameter values below the defined threshold to 0. I changed this algorithm by thresholding out the parameter values based on the relevant signal's energy, rather than the parameter value itself. The energy is given by both the magnitude of the respective data column vector and the parameter value. Alternatively, the original STLS method can be applied directly, if the data columns are energy normalized. As I've shown, multiplying the column vector by the inverse square root of the vector's energy normalizes the vector's total energy to 1.

The SINDy method was first tested on two simulated Lorenz systems. In the first case, the Lorenz system had additional external inputs, which entered the dynamics nonlinearly, for example through the sign function $\mathrm{sgn}(u)$. To simulate real measurements, white noise was added to the state measurements after simulation. These measurements were then filtered using spectral filtering, and state derivatives were estimated from the filtered state measurements using spectral differentiation. The discovered model was fairly close to the reference model used to generate the data, despite the imperfections in the training data. The other Lorenz system had inputs entering the dynamics linearly, but the inputs were defined by state feedback during the simulation. This causes perfect correlations between the state and input signals. These correlations have to be broken by adding a random signal to the input definition. Even when the system was feedback-controlled, the method successfully distinguished between the effects of natural dynamics and external forcing and found the correct model from data.

The next experiments consisted of identifying the model of a simulated pendulum-cart system from data using the SINDy-PI method. The analytical model was first derived using Euler-Lagrangian mechanics, and then used to generate the simulation data. The cart position and pendulum angle measurements were then differentiated to estimate the respective velocities, and then differentiated again to estimate the accelerations. Numerical differentiation is very sensitive to noise, which is a big problem when it's performed twice in a row. To remedy this problem, I found it's a good idea to over-filter the signal - instead of picking the filter cutoff frequency as the frequency at which the signal becomes weaker than the noise, it's picked as roughly the frequency at which the spectral signal-to-noise ratio starts dropping. This filtering step significantly improves the derivative estimation accuracy.

The most difficult part of the identification process is correctly defining the candidate function library. I found a somewhat systematic way to approach this problem. First, define a set of simple basis functions. In the pendulum-cart system's case, these were picked as the velocities, sine and cosine of the pendulum angle and the accelerations. The complete function library is then constructed as all possible combinations of those basis terms up to a given order. The resulting candidate functions are then evaluated using a set of manually created rules to discard the candidate functions that don't make physical sense. I also found that using the correlation matrix of the function library is a good way to quickly visually evaluate the library and identify potentially problematic candidate functions.

Because the SINDy-PI generates a large number of sparse models, I had to create a systematic way of picking the viable ones. The first, more obvious, step is discarding all duplicate models. Implementationally, this was done by calculating the hash function for each model, sweeping through all the models and keeping only models that have a hash which hasn't already been seen during the sweep. The next step is less obvious, and uses the fact that models with the correct structure should appear more frequency in the set of identified models. First, the model parameter vectors are transformed into the so called activation vectors. Then, a clustering algorithm is used to find clusters of models in the activation space. The idea is that models close to each other in the activation space have the same active parameters, so large clusters likely consist of good models.

The SINDy-PI method accurately identified the rational ODEs defining the system's cart and pendulum's accelerations. To validate the parameters, I used the technique known as bootstrapping to again generate 25000 models for each of the accelerations, using a function library containing only the active terms. The SINDy-PI definition of the regression task looks for implicit solutions; one of the columns of the function library is extracted and acts as the target variable (guess function) during regression. By picking this guess function randomly for each one of the models, I generate the parameter distributions for the model. These parameter distributions were sometimes highly polymodal, which is indicative of imperfect structure of the identified model.

As the last experiment, I tried identifying the model of a real physical system from its measurements. Unfortunately, due to very noisy and 0-shifted input measurements $\mathbf{U}$, the discovered model's derivative estimations were noisy. Despite this, the identified active terms were fairly accurate and at least partially resembled the analytically derived model. When evaluating the model "visually" using a numerical simulation, the qualitative behaviour of the model was believeable.

Acknowledgement

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS19/158/OHK2/3T/12.

Nomenclature

x_1	Cart position	m
x_2	Pendulum angle	rad
x_3	Cart velocity	$\mathrm{m\,s^{-1}}$
x_4	Pendulum angular velocity	$\mathrm{rad\,s^{-1}}$
$\ddot{x}_1\,\dot{x}_3$	Cart acceleration	$\mathrm{m\,s^{-2}}$
$\ddot{x}_2\,\dot{x}_4$	Pendulum angular acceleration	$\mathrm{rad\,s^{-2}}$
$\mathbf{x}$	State vector	
$\dot{\mathbf{x}}$	State derivative vector	
$\mathbf{\Theta}$	Candidate function library	
θ	Candidate function	
$\boldsymbol{\theta}$	Candidate function measurement vector	
$\boldsymbol{\xi}$	Vector of model parameters	
$\mathbf{a}$	Activation vector	
$\boldsymbol{X}$	Matrix of state measurements	
$\dot{\boldsymbol{X}}$	Matrix of state derivative measurements	
$\boldsymbol{U}$	Matrix of input measurements	

ODE	Ordinary Differential Equation
SINDy	Sparse Identification of Nonlinear Dynamics
SINDy-PI	SINDy - Parallel Implicit

Github link

All the code used in this thesis is available on my Github page: https://github.com/BystrickyK/SINDy/

References

[1] 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*, vol. 113, no. 15, pp. 3932–3937, Apr. 12, 2016, Publisher: National Academy of Sciences Section: Physical Sciences, ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.1517384113. [Online]. Available: https://www.pnas.org/content/113/15/3932 (visited on 03/31/2021).

[2] (). "Definition of PARSIMONY", [Online]. Available: https://www.merriam-webster.com/dictionary/parsimony (visited on 05/10/2021).

[3] ——, "Sparse identification of nonlinear dynamics with control (SINDYc)", *arXiv:1605.06682 [math]*, May 21, 2016. arXiv: 1605.06682. [Online]. Available: http://arxiv.org/abs/1605.06682 (visited on 03/31/2021).

[4] N. M. Mangan, S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Inferring biological networks by sparse identification of nonlinear dynamics", *arXiv:1605.08368 [math]*, May 26, 2016. arXiv: 1605.08368. [Online]. Available: http://arxiv.org/abs/1605.08368 (visited on 04/15/2021).

[5] K. Kaheman, J. N. Kutz, and S. L. Brunton, "SINDy-PI: A robust algorithm for parallel implicit sparse identification of nonlinear dynamics", *arXiv:2004.02322 [physics, stat]*, Sep. 29, 2020. arXiv: 2004.02322. [Online]. Available: http://arxiv.org/abs/2004.02322 (visited on 03/31/2021).

[6] E. Kaiser, J. N. Kutz, and S. L. Brunton, "Sparse identification of nonlinear dynamics for model predictive control in the low-data limit", *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 474, no. 2219, p. 20 180 335, Nov. 30, 2018, Publisher: Royal Society. DOI: 10.1098/rspa.2018.0335. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rspa.2018.0335 (visited on 05/17/2021).

[7] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton, "Data-driven discovery of coordinates and governing equations", *Proceedings of the National Academy of Sciences*, vol. 116, no. 45, pp. 22 445–22 451, Nov. 5, 2019, Publisher: National Academy of Sciences Section: Physical Sciences, ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.1906995116. [Online]. Available: https://www.pnas.org/content/116/45/22445 (visited on 05/21/2021).

[8] G. C. Goodwin, S. F. Graebe, and M. E. Salgado, *Control System Design*. Upper Saddle River, NJ: Pearson, Oct. 6, 2000, 944 pp., ISBN: 978-0-13-958653-8.

[9] J. Dormand and P. Prince, "A family of embedded runge-kutta formulae", *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, Mar. 1980, ISSN: 03770427. DOI: 10.1016/0771-050X(80)90013-3. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0771050X80900133 (visited on 10/23/2020).

[10] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: Symbolic computing in python", *PeerJ Computer Science*, vol. 3, e103, Jan. 2017, ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`. [Online]. Available: `https://doi.org/10.7717/peerj-cs.103`.

[11] R. Tibshirani, "Regression shrinkage and selection via the lasso", *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996, ISSN: 00359246. [Online]. Available: `http://www.jstor.org/stable/2346178`.

[12] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge, UK ; New York: Cambridge University Press, 2004, 716 pp., ISBN: 978-0-521-83378-3.

[13] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems", *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970. DOI: `10.1080/00401706.1970.10488634`. eprint: `https://www.tandfonline.com/doi/pdf/10.1080/00401706.1970.10488634`. [Online]. Available: `https://www.tandfonline.com/doi/abs/10.1080/00401706.1970.10488634`.

[14] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, 1st ed. Cambridge University Press, Jan. 31, 2019, ISBN: 978-1-108-38069-0 978-1-108-42209-3. DOI: `10.1017/9781108380690`. [Online]. Available: `https://www.cambridge.org/core/product/identifier/9781108380690/type/book` (visited on 03/31/2021).

[15] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy", *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: `10.1038/s41586-020-2649-2`. [Online]. Available: `https://doi.org/10.1038/s41586-020-2649-2`.

[16] (). "Scipy.signal.convolve — SciPy v1.6.0 reference guide", [Online]. Available: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.html` (visited on 01/03/2021).

[17] P. Welch, "The use of fast fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms", *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967. DOI: `10.1109/TAU.1967.1161901`.

[18] H. Akaike, "A new look at the statistical model identification", *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, Dec. 1974, ISSN: 0018-9286. DOI: `10.1109/TAC.1974.1100705`. [Online]. Available: `http://ieeexplore.ieee.org/document/1100705/` (visited on 04/20/2021).

[19] K. P. Burnham, D. R. Anderson, and K. P. Burnham, *Model selection and multimodel inference: a practical information-theoretic approach*, 2nd ed. New York: Springer, 2002, 488 pp., OCLC: ocm48557578, ISBN: 978-0-387-95364-9.

[20] S. Kullback and R. A. Leibler, "On information and sufficiency", *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, Mar. 1951, Publisher: Institute of Mathematical Statistics, ISSN: 0003-4851, 2168-8990. DOI: 10.1214/aoms/1177729694. [Online]. Available: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-1/On-Information-and-Sufficiency/10.1214/aoms/1177729694.full (visited on 08/02/2021).

[21] T. Glück, A. Eder, and A. Kugi, "Swing-up control of a triple pendulum on a cart with experimental validation", *Automatica*, vol. 49, no. 3, pp. 801–808, Mar. 2013, ISSN: 00051098. DOI: 10.1016/j.automatica.2012.12.006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S000510981200605X (visited on 04/11/2021).

[22] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. Boston, MA: Springer US, 1993, ISBN: 978-0-412-04231-7 978-1-4899-4541-9. DOI: 10.1007/978-1-4899-4541-9. [Online]. Available: http://link.springer.com/10.1007/978-1-4899-4541-9 (visited on 07/31/2021).

[23] (). "Triple inverted pendulum IPM-310", REX Controls, [Online]. Available: https://www.rexcontrols.com/triple-inverted-pendulum-ipm-310/ (visited on 08/01/2021).