School of Mathematics

MSc Dissertation 2022/2023

Plagiarism Declaration Form

Student Name: OSKAR LJUNGDELL

Student ID Number: 2497501

Dissertation Title: TWO-STAGE FILTERS FOR IMPROVED IDENTIFICATION OF NONLINEAR GOVERNING DYNAMICS FROM DATA IN THE HIGH NOISE LIMIT.

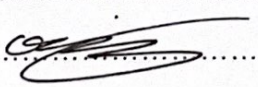
Please see the School of Mathematics Canvas Page on Academic Integrity:

https://canvas.bham.ac.uk/courses/65757/pages/academic-integrity

Please E-mail Dr David Leppinen, Director of MSc Programmes, (D.M.Leppinen@bham.ac.uk) if you have any queries.

A signed copy of this Plagiarism Declaration Form must be submitted electronically along with your Dissertation to confirm that you agree to the following: "I warrant that the content of this Dissertation is the direct result of my own work and that any use made in it of published or unpublished material is fully and correctly referenced." Students will receive a Dissertation Mark of 0 if they do not complete this form. The same Penalties for late submission of this Plagiarism Declaration Form will apply as for the late submission of the actual Dissertation.

Name: OSKAR LJUNGDELL

Signature: 

University of Birmingham

School of Mathematics

MSc Applied Mathematics Dissertation

Two-stage filters for improved identification of nonlinear governing dynamics from data in the high noise limit

---

Author:

Oskar Ljungdell

Supervisors:

Dr. Fabian Spill

Dr. Daniel Herring

October 3, 2023

Abstract

In this study, we explore the use of a priori filtering techniques as pre-processing steps to improve the successful identification of nonlinear dynamics from data corrupted by Gaussian noise. Specifically, we apply these under the adoption of the recently proposed weak formulation of the popular sparse regression method known as the sparse identification of nonlinear dynamics (SINDy) algorithm. We investigate the local Savitzky-Golay and global ℓ_1 filters with Pareto tuned hyperparameters in reconstructing the Lorenz ordinary differential equation (ODE) system and viscous Burgers' partial differential equation (PDE) over varying noise strengths. Additionally, through qualitative findings of distortions seen by singular filters in high noise bands, we propose the novel design of using a two-stage filter to smooth out the inconsistencies generated by a single filter. Recognizing the variation in filtering designs, we look into two combinations: local followed by global and vice versa. Performing experimental tests on these systems using these various designs, we find that below the Gaussian noise range of 15‰-20‰, filtering damages the ability to construct the system correctly. Above this range, however, both minor and major benefits are observed. In the ODE setting, although the successful identification rate is not significantly altered, coefficient errors, and particularly so for correct identifications, get reduced. This indicates the inherent strength of the weak formulation, but we proceed to show that this enhancement is still helpful. In the PDE setting, these improvements are heightened with both success rates and coefficient errors seeing a large increase in performance. It is also found that each of the two-stage filtering designs are able to further robustify their singular first components across the range of error metrics tested, motivating the adoption of this strategy for systems that are described by data estimated to contain a higher amount of noise than the indicated threshold.

Acknowledgements

In completing this dissertation, I wish to extend my heartfelt gratitude to Dr. Fabian Spill for introducing me to this fascinating field and for providing guidance in selecting my dissertation topic. Furthermore, I owe profound appreciation to Dr. Daniel Herring whose invaluable assistance enabled my code to run on the university's high-performance supercomputer, leading to the major experimental results presented herein. Last but not least, I want to express the deepest thanks to my family for their mental support throughout this journey.

Contents

Notations

- **a** : A vector is represented by a bold lowercase letter.
- **M** : A matrix is represented by a bold uppercase letter.

In the following definitions, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is a vector.

- $\|\mathbf{x}\|_0$: The ℓ_0 pseudo-norm of a vector **x** represents the number of non-zero elements in the vector. If $\chi(x_i)$ is an indicator function that is 1 when $x_i \neq 0$ and 0 otherwise, then:

$$\|\mathbf{x}\|_0 = \sum_{i=1}^{n} \chi(x_i).$$

- $\|\mathbf{x}\|_1$: The ℓ_1 norm of a vector **x**, also known as the Manhattan norm, represents the sum of the absolute values of its elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|.$$

- $\|\mathbf{x}\|_2$: The ℓ_2 norm of a vector **x**, also known as the Euclidean norm, represents the square root of the sum of the squares of its elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

1 Introduction

1.1 Motivation

In recent times, there have been significant advancements in our ability to collect, process, and store massive data streams in fields such as engineering, finance, and neuroscience. With this, we have been able to attain large libraries of information that describe how many systems function, yet their governing dynamics remain unknown. The interest in applying data driven approaches to extract interpretable models, or these governing dynamics, from such measured data has only surged over time. A plethora of methods for data driven model discovery exist at present time, all aimed at avoiding the need of a derivation from first principles. Some of these include making use of symbolic regression [1], deep learning [2], dynamic mode decomposition [3], and Koopman analysis [4]. A major proposal that has shown a lot of promise is the sparse identification of nonlinear dynamics, or SINDy, algorithm [5]. Being founded on a sparse linear regression approach, it aims to yield discovered dynamics of the sparse and interpretable form we are after. Moreover, it also requires significantly less data to operate than methods like neural networks [6], allowing for less computational overhead. However, being innately based on the input data, all of these approaches suffer from the same core weakness deeply rooted in the field of data science, namely sensitivity to the presence of noise disruption.

When collecting real-world measured data, noise is able to permeate on many fronts. First and foremost, it is a foundational principle in engineering that all sensors we use to measure information have some form of intrinsic error associated with them. This is an unavoidable fact, but advances in technology generally limit these errors to a very small amount in most situations. From an engineering perspective, well calibrated and accurate sensors would still produce a high signal-to-noise ratio (SNR) in the presence of no other large noise sources, and in this scenario inference from the data would still be straightforward. This, however, is not always the case. Furthermore, aside from the intrinsic noise present in sensors, certain dynamical systems exhibit stochastic features that are fundamental in how they evolve with time. For instance, the firing

of brain neurons is not fully deterministic due to a multitude of reasons such as being impacted by various external factors like neurotransmitter concentrations [7]. Similarly, mathematical models of financial markets generally apply Brownian motion to model the random fluctuations in price [8], with elements such as the participation of traders in the market also being inherently random. In these cases when the goal is to decipher the overall governing dynamics of the system, these random fluctuations ultimately become a form of irreducible noise as well.

Therefore, in real world problems where a currently unknown governing dynamic is trying to be found, the issue of noise corruption is inevitably present. Many studies have investigated techniques like feature engineering [9], ensembling [6], and even denoising filters for the original SINDy implementation [10] to reduce the algorithm's sensitivity to noise. Since much of this work and particularly that on filtering had been done, a more noise robust implementation of SINDy known as the weak formulation has been introduced, and is currently the recommended algorithm to study noisy systems [11]. However, being a recent publication, it is unclear whether previously tested methods are still helpful given that it uses a different core approach. To this end, our study is centered around finding out whether a priori filtering is still helpful in the new context of applying the weak formulation. We also propose the design of using a stacked two-stage filter and investigate this as a means of handling even higher levels of noise, with the goal of conclusively being able to see the effectiveness of filters in this setting and thus its potential usefulness in tackling noise heavy data.

1.2 Definition of a Dynamical System

A dynamical system is a system whose state evolves with time according to a fixed rule. More formally, it consists of a state space with coordinates that describe the state at any time, together with a dynamical rule. This dynamical rule, when given present values of the state variables, is able to specify the immediate future of all said variables. Therefore, dynamical systems provide a framework to describe anything around us that evolves with time. In a general sense, we can model a dynamical system as follows:

$$\frac{d}{dt}\mathbf{x} = \boldsymbol{f}(\mathbf{x}, \boldsymbol{\beta}, t), \tag{1}$$

where $\mathbf{x}$ is the state vector, $\boldsymbol{\beta}$ is a vector of parameters that describe the specific system, t is the time, and $\boldsymbol{f}$ is the dynamical rule or a set of functions $(f_1, f_2, f_3, \dots)$ that describe the dynamics of states $(s_1, s_2, s_3, \dots)$.

For most systems of actual interest, their corresponding dynamics $\boldsymbol{f}$ tend to be of a nonlinear form. This means that its evolution in time cannot simply be described by linear functions of its states. Rather, we require more intricate base components such as higher powers than 1 of the state variables, interaction terms between them, or other nonlinear mathematical functions like trigonometric and exponential ones. This naturally leads to nonlinear dynamical systems being able to exhibit a range of complex behaviours that are not trivially present in linear ones. For instance, an observation of particular interest is that of chaos and bifurcations, which is when a small change in a parameter in $\boldsymbol{\beta}$ results in a qualitative change in the long-time solution [12].

The major challenge in modern and highly complex systems is that we do not actually know the governing dynamics $\boldsymbol{f}$. This is because the system at hand simply does not have a known derivation from first principles to describe its governing dynamics, and hence why data driven methods to just infer $\boldsymbol{f}$ has seen so much study. A key goal as well in obtaining a describing model for a dynamical system is for this model to be generizable and interpretable. In other

words, we want a model in as simple of a form as possible that still captures the wanted dynamics, allowing for it to be easily analyzed and extended to other scenarios. Ultimately, this is why the core framework of SINDy is so popular given its sparsity promoting approach. However, going back onto the topic of bifurcations, it is also very clear that the discovered dynamics do not just have to contain the correct terms, but that their corresponding coefficients also have to be as close to the true values as possible. If we are dealing with a chaotic system and this fails, being especially common with noise corrupted data, our discovered dynamics would deviate from the true form as enough time passes. Therefore, the interest in being able to augment the algorithm's performance by any means to avoid such an unwanted pitfall is abundantly clear.

2 Literature Review

2.1 Sparse Identification of Nonlinear Dynamics Algorithm

The sparse identification of nonlinear dynamics, or SINDy, algorithm introduced by Steven Brunton et al. [5] has garnered much attention in the scientific community due to its emphasis on promoting sparsity. Its inherent design ensures that SINDy identifies the simplest possible model that adequately represent the underlying dynamics, or in other words a parsimonious model corresponding to the desired interpretable representation. For the originally targeted ordinary differential equation (ODE) setting, consider a general nonlinear dynamical system given by:

$$\frac{d}{dt}\mathbf{x}(t) = \boldsymbol{f}(\mathbf{x}(t)), \tag{2}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state vector of the system at time t and $\boldsymbol{f}$ defines the dynamics of the system.

The SINDy algorithm relies on the assumption that $\boldsymbol{f}$ has relatively few active terms, meaning that it is sparse when compared to all possible functions that could model $\mathbf{x}(t)$. Provided we have data of n measured variables stored in a matrix $\mathbf{X}$ representing the system sampled at m equidistant time intervals, we can apply numerical differentiation like finite differences to compute approximate time derivatives. This allows us to also construct the matrix $\dot{\mathbf{X}}$, meaning that we have:

$$\mathbf{X} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \dots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \dots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \dots & x_n(t_m) \end{bmatrix} \quad \text{and} \quad \dot{\mathbf{X}} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \dots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \dots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \dots & \dot{x}_n(t_m) \end{bmatrix},$$

where we make use of the classical dot notation format $\dot{()}$ to represent the approximated time derivatives.

The next step is to construct a suitable library of C candidate functions $\boldsymbol{\Theta}(\mathbf{X}) \in \mathbb{R}^{m \times C}$ that we are later going to apply regression on:

$$\boldsymbol{\Theta}(\mathbf{X}) = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \mathbf{X} & \mathbf{X} & \dots & \sin(\mathbf{X}) & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}.$$

The choice of candidate functions here is critical, and especially difficult if the true dynamics of the system is unknown. This is because the library must include the forms of terms that are ultimately present in the dynamics $\boldsymbol{f}$ we are trying to find. It is important to note however that the number of possible models from this library is combinatorially large, and that polynomials tend to be of special interest since they are often key parts in canonical models of dynamical systems [13]. In the end though, if the dynamics are unknown this turns into a problem specific task where general knowledge of the system is helpful. For instance, if it is known that waves constitute the dynamics of a similar problem in an easier class it would be natural to include trigonometric functions in the candidate library.

Having defined these data matrices, we can now use them to reformulate our general expression of the dynamical system in eq. (2) as follows:

$$\dot{\mathbf{X}} = \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\Xi}, \tag{3}$$

where $\boldsymbol{\Xi} = (\boldsymbol{\xi}_1, ..., \boldsymbol{\xi}_n) \in \mathbb{R}^{C \times n}$ are the respective coefficient vectors indicating the active terms from the candidate library in each row of the discovered dynamical system. The notion of multiple rows here refer to the case where the governing system contain time derivatives for each respective dimension, or in other words it being multidimensional. If this is not the case, there would just be a single row.

This has now resulted in an optimization problem to find a sparse $\boldsymbol{\Xi}$ that represents the data in $\dot{\mathbf{X}}$. We obtain a parsimonious model by performing regression on this system with a sparsity promoting regularizer $R(\hat{\boldsymbol{\Xi}})$:

$$\boldsymbol{\Xi} = \underset{\hat{\boldsymbol{\Xi}}}{\operatorname{argmin}} \left(\frac{1}{2} \left\| \dot{\mathbf{X}} - \boldsymbol{\Theta}(\mathbf{X}) \hat{\boldsymbol{\Xi}} \right\|_2^2 + R(\hat{\boldsymbol{\Xi}}) \right). \tag{4}$$

It is important to note that there exists a wide choice of regularizers $R(\hat{\boldsymbol{\Xi}})$. For ODE systems, the approach of sequentially thresholded ridge regression (often referred to as sequentially thresholded least squares with ridge regularization) tends to be a popular choice [6]. The regularization in this case is expressed as $R(\hat{\boldsymbol{\Xi}}) = \lambda \left\| \hat{\boldsymbol{\Xi}} \right\|_2^2$ [14], where λ represents the regularization strength. This means that one ultimately has to solve:

$$\boldsymbol{\Xi} = \underset{\hat{\boldsymbol{\Xi}}}{\operatorname{argmin}} \left(\frac{1}{2} \left\| \dot{\mathbf{X}} - \boldsymbol{\Theta}(\mathbf{X}) \hat{\boldsymbol{\Xi}} \right\|_2^2 + \lambda \left\| \hat{\boldsymbol{\Xi}} \right\|_2^2 \right). \tag{5}$$

The original SINDy algorithm performs a slight variation of this known as STLSQ, which is what we adopt as our implementation as well. It takes in an additional parameter η that specifies the minimum magnitude for a coefficient in $\boldsymbol{\Xi}$. Following a least squares fit, the idea is then to zero out all coefficients with a magnitude below this value η. This process is repeated until a specified level of convergence is reached [15].

After the sparse set of coefficients $\boldsymbol{\Xi}$ have been determined, the discovered models for each row of the governing equations in eq. (2) can be constructed as follows, noting that $\boldsymbol{\Theta}(\mathbf{x}^T)$ is a vector of symbolic functions of elements of $\mathbf{x}$ [5]:

$$\dot{\mathbf{x}}_k = \boldsymbol{f}_k(\mathbf{x}) = \boldsymbol{\Theta}(\mathbf{x}^T) \boldsymbol{\xi}_k. \tag{6}$$

2.2 SINDy Extended to PDE Systems

Following the proposal of SINDy, it was quickly generalized by Samuel Rudy et al. [16] to include the functionality of being able to identify partial differential equation (PDE) systems as well. Their specific implementation is known as the partial differential equation functional identification of nonlinear dynamics, or PDE-FIND, algorithm, but only varies from the original SINDy framework by including partial derivatives in the candidate library. To showcase the distinction, we consider nonlinear PDEs of the form:

$$\mathbf{u}_t = \mathbf{F}(\mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}, \ldots, x, \mu), \tag{7}$$

where $\mathbf{F}(\cdot)$ is the system of generally nonlinear functions of the state $\mathbf{u}(x,t)$, its derivatives, and any parameters μ that we aim to determine. Subscripts denote partial differentiation in time or space, such that $\mathbf{u}_x := \partial\mathbf{u}/\partial x$. Although we focus on a single dependent variable $\mathbf{u}(x,t)$ here for illustratory simplicity, this algorithm is capable of handling systems with multiple dependent variables just like SINDy for the ODE case as detailed in the previous subsection. When approximating derivatives, the PDE-FIND implementation provides a choice in whether to use regular finite differences if the data is known to be clean, or polynomial interpolation if it noisy [16]. This essentially corresponds to an a priori built in filter.

In this framework, as we are dealing with partial differential equations, we require time series data over m time points and n spatial locations for the variable $\mathbf{u}(x,t)$. This gives us a state matrix of form $\mathbf{U} \in \mathbb{R}^{mn}$ to work with. Furthermore, with the candidate library $\boldsymbol{\Theta}(\mathbf{U})$ now including partial derivatives as well, the amount of candidate functions C is considerably larger now than before in the ODE formulation. With this in mind, the entire library is of magnitude $\boldsymbol{\Theta}(\mathbf{U}) \in \mathbb{R}^{mn \times C}$, and we can represent this as:

$$\boldsymbol{\Theta}(\mathbf{U}) = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \mathbf{U} & \mathbf{U}^2 & \ldots & \mathbf{U_x} & \ldots & \mathbf{UU_x} & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}.$$

Following the same idea as before, we can now rewrite our PDE-based dynamical system in eq. (7) with the constructed data matrices and a sparse coefficient matrix $\boldsymbol{\Xi}$ where $\mathbf{U}_t$ captures the rate of change of the variable with respect to time at all measured points:

$$\mathbf{U}_t = \boldsymbol{\Theta}(\mathbf{U})\boldsymbol{\Xi}. \tag{8}$$

With this setup, we again arrive at the optimization problem of finding a sparse $\boldsymbol{\Xi}$ that captures our data but now in $\mathbf{U}_t$. The regression framework remains the same where we have to choose a sparsity promoting regularizer $R(\hat{\boldsymbol{\Xi}})$:

$$\boldsymbol{\Xi} = \underset{\hat{\boldsymbol{\Xi}}}{\operatorname{argmin}} \left(\frac{1}{2} \left\| \mathbf{U}_t - \boldsymbol{\Theta}(\mathbf{U})\hat{\boldsymbol{\Xi}} \right\|_2^2 + R(\hat{\boldsymbol{\Xi}}) \right). \tag{9}$$

Like earlier, there is once more a large range of regularizers to pick from. For PDE systems where data can exhibit strong correlations temporally and spatially in particular measurement regions, the STLSQ approach still works well [6]. However, due to the larger candidate libraries and data matrices to account for both time and space in this problem formulation, another regularizer known as sparse relaxed regression (SR3) has been developed for increased computational efficiency and accuracy [17]. Therefore, we adopt this instead when working in the PDE setting. The idea is that we introduce an auxiliary variable $\mathbf{W}$ that approximates $\hat{\boldsymbol{\Xi}}$, and then relax the optimization problem as follows to decouple the regularization of $\hat{\boldsymbol{\Xi}}$:

$$\boldsymbol{\Xi} = \underset{\hat{\boldsymbol{\Xi}},\mathbf{W}}{\operatorname{argmin}} \left(\frac{1}{2} \left\| \mathbf{U}_t - \boldsymbol{\Theta}(\mathbf{U})\hat{\boldsymbol{\Xi}} \right\|_2^2 + \lambda R(\mathbf{W}) + \frac{1}{2\nu} \left\| \hat{\boldsymbol{\Xi}} - \mathbf{W} \right\|_2^2 \right), \tag{10}$$

where ν is a relaxation hyperparameter that modulates the gap between $\hat{\boldsymbol{\Xi}}$ and $\mathbf{W}$, and $R(\mathbf{W})$ denotes the regularization function. By instead focusing the regularization on $\mathbf{W}$, we can employ nonconvex regularization functions like the ℓ_0 pseudo-norm which represents hard thresholding [15]. This eliminates the necessity for an extra thresholding step and is thus what we utilize. It is important to note that $\mathbf{W}$ is not specified but dynamically determined throughout the optimization process. Once the sparse set of coefficients $\boldsymbol{\Xi}$ has been found, the discovered dynamics can be extracted in the same way as for the ODE formulation as shown by eq. (6).

2.3 Weak Formulation of SINDy

The initial SINDy frameworks presented thus far contain a fundamental flaw when it comes to robust performance within a noisy setting. Specifically, they require numerical approximations of the derivatives in $\dot{\mathbf{X}}$ and $\mathbf{U}_t$. The simplest way to approach this is by using a standard finite difference (FD) scheme, and this works well if no noise is present. However, when noise is present, most numerical differentiation methods, this one included, is prone to greatly amplify this noise [18]. To diminish the extent of this, the original implementation suggested the use of total variation (TV) regularization for the approximations if the data is known to be noisy [5].

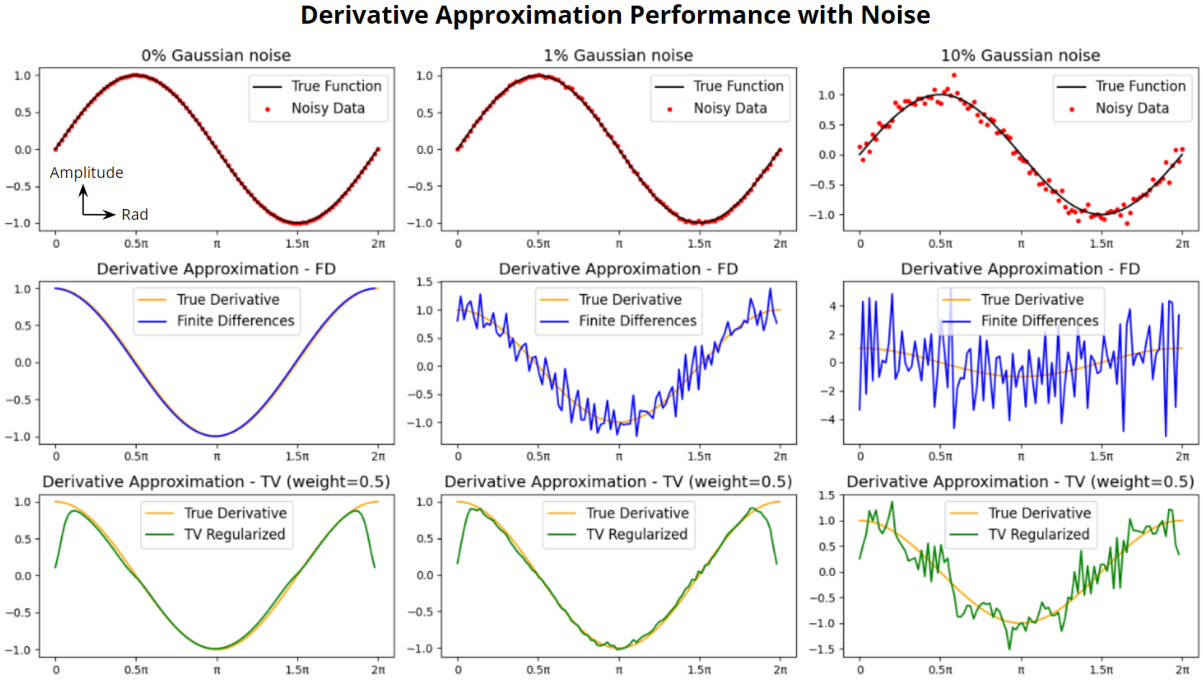


Figure 1: Performance of finite differences (FD) and total variation (TV) regularization methods to approximate the derivative of $sin(x)$ under Gaussian noise addition of 0, 1, and 10 percent.

By plotting the performance of both the finite difference and total variation approach across a few noise addition percentages, we can see that the finite difference scheme immediately starts to break down already at 1% of added Gaussian noise. The total variation method, on the other hand, is able to follow the true derivative more closely, but starts to see similar disruptions at 10% noise where the finite difference method has already totally failed. We note that although TV regularization is shown to help out a lot, it still leaves a significant amount to be desired.

The founding idea behind using a weak formulation is to transform the original problem of working with a differential equation into that of an integral equation instead. This is to avoid the shortcomings of using numerical methods to approximate derivatives [19], which as shown pose a particular issue in the presence of noise, by simply eliminating the need for the operation altogether. The first studied approach targeted at this was that of using integral terms [20], with a weak formulation for SINDy quickly following through work by Patrick Reinbold et al. [11] which is what we adopt here. It is important to note that since this publication other strategies for the weak formulation have been proposed [21, 22], but that we restrict our study to the initial formulation. To showcase the approach, we once again consider the nonlinear partial differential equation form for a dynamical system but also note the inclusion of an initial condition:

$$\mathbf{u}_t = \mathbf{F}(\mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}, \ldots, x, \mu), \quad \mathbf{u}(t = 0) = \mathbf{u}_0. \tag{11}$$

If we define a vector of basis functions $\mathbf{f}_n(\mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}, \ldots, x, \mu)$ with corresponding coefficients c_n, then the right hand side of the partial differential equation in eq. (11) can be reformulated using summation notation as follows:

$$\mathbf{u}_t = \sum_{n=1}^{N} c_n \mathbf{f}_n(\mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}, \ldots, x, \mu). \tag{12}$$

The plan now is to multiply this form of the differential equation by a weight $\mathbf{w}_j(\mathbf{u}, t)$ and then integrate it over a domain Ω_k. If this process is repeated over a large range of combinations of weights $\mathbf{w}_j(\mathbf{u}, t)$ and K domains Ω_k, then the results of integration can be stacked to arrive the following linear system:

$$\mathbf{q}_0 = \sum_{n=1}^{N} c_n \mathbf{q}_n, \tag{13}$$

where $\mathbf{q}_0$ is the integrated left hand side of the partial differential equation and $\mathbf{q}_n \in \mathbb{R}^K$ are the column vectors corresponding to different terms f_n with entries that depend on the particular choice of weight $\mathbf{w}$ and domain Ω_k [11]. The terms $\mathbf{q}_n$ are of the form:

$$q_n^k = \int_{\Omega_k} \mathbf{w} \cdot \mathbf{f}_n \, d\Omega. \tag{14}$$

The key importance of this formulation where the basis functions $\mathbf{f}_n$ contain the derivatives, is that the act of actual differentiation can be transferred from the potentially noisy data $\mathbf{u}$ onto the smooth weight $\mathbf{w}$ through integration by parts. To highlight this, consider the case where one of the terms $\mathbf{f}_n$ is just $\mathbf{u}_x$. We therefore have:

$$q_n^k = \int_{\Omega_k} \mathbf{w} \cdot \mathbf{u}_x \, d\Omega. \tag{15}$$

Applying integration by parts and having chosen a weight $\mathbf{w}$ such that the boundary terms vanish [11], we get:

$$q_n^k = -\int_{\Omega_k} \mathbf{w}_x \cdot \mathbf{u} \, d\Omega. \tag{16}$$

It is now clear that instead of needing a derivative approximation over the data $\mathbf{u}$, we instead only need to differentiate the smooth weight function $\mathbf{w}$. It is important to note that the performance limiting component in this setup however is the approximation of the definite integrals, which also requires a numerical scheme. Although multiple such schemes exist, the original implementation suggests the use of the composite trapezoidal rule [11]. Ultimately, we can rewrite the stacked linear system in eq. (13) in our standard format [6]:

$$\mathbf{q}_0 = \mathbf{Q}\boldsymbol{\Xi}, \tag{17}$$

where $\mathbf{Q} = [\mathbf{q}_1, ..., \mathbf{q}_N]$ is the integrated library of basis functions or candidate terms and $\boldsymbol{\Xi}$ the coefficient matrix indicating active terms. For $K \geq N$, this is an over-determined system with the number of equations in $\mathbf{Q}$ being larger than the number of unknowns in $\boldsymbol{\Xi}$. Therefore, we have now arrived at the same problem as before of finding a parsimonious model or a sparse $\boldsymbol{\Xi}$ that still captures the data, noting that we have completely avoided the need for any derivative approximations of $\mathbf{u}$. To find a sparse $\boldsymbol{\Xi}$, the problem is formulated just like before and is the baseline algorithm we adopt moving forwards, keeping in mind that we choose the regularizer $R(\hat{\boldsymbol{\Xi}})$ to be STLSQ for an ODE system and SR3 for a PDE system:

$$\boldsymbol{\Xi} = \underset{\hat{\boldsymbol{\Xi}}}{\operatorname{argmin}} \left(\frac{1}{2} \left| \mathbf{q}_0 - \mathbf{Q}\hat{\boldsymbol{\Xi}} \right|_2^2 + R(\hat{\boldsymbol{\Xi}}) \right). \tag{18}$$

2.4 Further work on Noise Robustness

Having already touched on this topic slightly in Section 1.1, with the core algorithms now defined it is possible to go into more depth. In the context of the up to date weak formulation, the main significant addition for further noise robustness that has been studied is the use of ensembling by Fasel et al. [6]. This is a machine learning technique which at a high level works by aggregating predictions from multiple models to produce a final output. In this work specifically, the popular method of boostrap aggregation, or bagging, was investigated both in terms of creating multiple datasets and multiple candidate libraries by sampling from the original datasets with replacement. In order to aggregate the results from all of the trained models, the mean of the identified coefficients was used. It was found that this approach not only helped the weak implementation perform better in a highly limited data setting, which is what generally motivatves the use of ensembling, but it did in fact improve tolerance to noise as well.

Another technique that has been investigated, albeit still under the original SINDy implementation, is that of using feature engineering as a preprocessing step. This was work done by França et al. [9] and was also found to increase SINDy's robustness to higher noise levels. The main idea behind feature engineering is to eliminate measured data features that are incapable of actually representing the true dynamics. It was found that even in cases where actual measurements cannot be discarded, using a dimensionality reduction method like principal component analysis (PCA) or singular value decomposition (SVD) could transform noisy, correlated data into a new set of features with less correlation that the SINDy algorithm handled better.

In terms of a priori filter denoising, this has also been studied but still only under the original SINDy framework. This was done by Cortiella et al. [10] and was motivated to tackle the issue of noise amplification when derivative approximations were still needed. Having investigated techniques both under the local and global filtering domain, they saw improvements from both but highlighted that global methods tended to have a slight edge. Ultimately, they noted that it would be of interest to extend the study to that of weak or integral formulations as well.

2.5 Our Contribution

With it being unclear at present time whether using filtering as a pre-processing step is still helpful under the framework of the weak SINDy formulation, we first seek to find this out. We investigate the application of the local Savitzky-Golay [23] and global ℓ_1 trend [24] filters in this weak setting for increasing levels of Gaussian noise corruption. Furthermore, with the goal of wanting to achieve further noise robustness, we also propose the novel design in this realm of using a stacked two-stage filter by combining the local and global ones. We explore both the local-global and global-local configuration setups alongside the standalone filters across a range of error metrics for identification of the canonical Lorenz system (ODE) and viscous Burgers' equation (PDE). In order to obtain valid tests for these approaches, we first apply Pareto L-curve criterion tuning [25] in selecting the most optimal filter hyperparameters before any application. This is to avoid under or over-smoothing for noise levels that do not align with a manually chosen fixed parameter, as well as to ensure that the applied filter(s) perform as good as possible throughout every level tested without assumed knowledge of the noise structure.

3 Denoising Filters

3.1 Local and Global Methods

Denoising filters can be categorized into two main areas depending on their scope of their operation: local and global methods. This distinction is made solely on the basis of the frame of data that these techniques work on, rather than the specific mathematical or algorithmic construction of the operation.

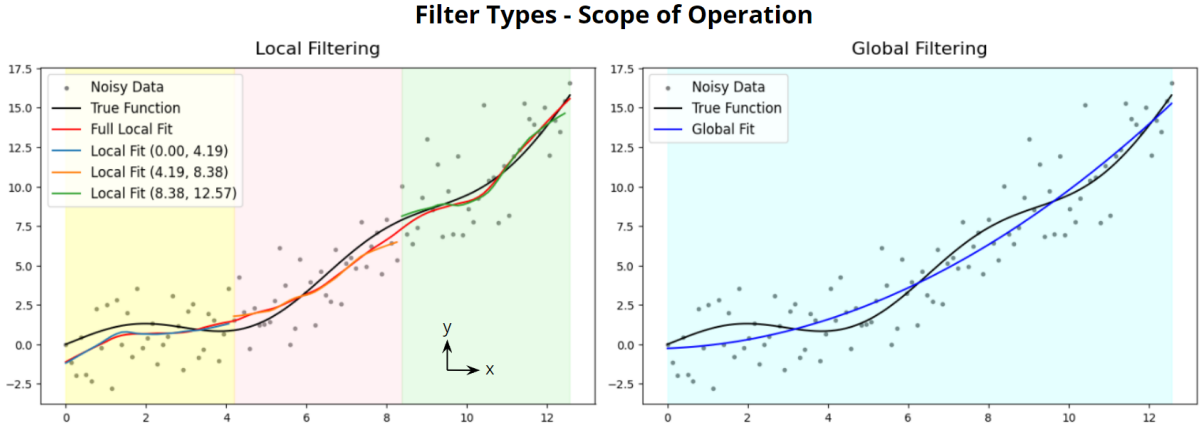


Figure 2: Scope of operation of local and global filtering methods

Local techniques operate by analyzing a specified neighborhood around each data point, generating individual fits for each region, and then consolidating these fits into a single filtered signal. One significant advantage of local methods is their ability to retain critical features in the data while effectively eliminating much of the noise [23]. In the context of digital image processing, such critical features could include sharp discontinuities or distinct edges and corners. This retention capability stems from the fact that the fittings are deliberately restricted to smaller subsets of data. However, it's important to note that choosing an excessively small neighborhood would lead to over-fitting to the noise itself. In contrast, global methods consider the entire data range at once in performing their operations [24]. This means they have a holistic view of the data, which allows them to enforce smoothness over larger areas. While this makes up for the possible pitfall of fitting more to noise in a local approach, we note an equivalent problem of inevitably filtering away more finer details here. Therefore, a clear trade off exists.

3.2 Savitzky-Golay Filtering (Local)

First introduced by Abraham Savitzky and Marcel Golay in 1964 [23], the Savitzky-Golay filter remains to this day a highly popular smoothing filter within signal processing. It is built on the premise of using polynomial regression on each data segment, where we will refer to this chosen data segment length as the window size, denoted by W. Due to the more exotic fits possible, this type of approach has been found to further help with feature preservation than more basic methods like moving window averaging [26], which simply works by combining the results from the mean of all data points in each window. To illustrate how the Savitzky-Golay filter works, we present the core framework below:

Algorithm 1 Savitzky-Golay Filtering Overview

procedure SAVITZKYGOLAY
    Define window $W = 2M + 1$ for samples $x(-M), \ldots, x(M)$
    Define polynomial degree n for interpolatory fit
    for $t = 0$ **to** T **do**
        Shift window to samples $x(t - M), \ldots, x(t + M)$
        if Samples exceed measured data **then**
            Pad empty cells
        end if
        Compute the least squares polynomial fit of degree n
        Compute the filter output $y(t)$ as the center value of the polynomial fit
    end for
end procedure

The first important note to make is in regards to the required parameter selections: window size W and polynomial degree n. Both of these parameters influence the extent of smoothing applied to the data. A larger window size W, where more data points are considered, often leads to under-parametrization of the regression function and thus increases the estimation bias due to curvature effects [27]. Likewise, as mentioned in the previous section, if the window size W is restricted to being too small, little smoothing would take place as we would over-fit to the noise giving a higher variance. In a similar fashion, a high polynomial degree n would allow for the capture of more features, noting that an exceedingly high degree would once again just over-fit to the features of the noise as well. Like a larger window size W, a lower polynomial order n

will promote smoothing, noting that too small of a degree n would also increase the bias due to curvature effects. For both it is clear that some form of optimal value exists. However, with our ultimate goal being to use this to try and uncover the dynamics of a system from noise corrupted measurements, we will use a fixed degree n guided by the highest derivative order we need to reconstruct [28]. This is the same approach used by Cortiella et al. [10] in their study, and we note that this is the only part of our investigation where some known information about the true system is utilized. In a completely unknown setting, a range of degrees n would have to be tested to find a setting that maximizes a metric like sparsity and consequently interpretability. In regards to the optimal choice of W, this depends on how much noise is present and is discussed in Section 3.4.

A further critical point to highlight is that the choice of the window size W must be odd. This is to ensure symmetry by allowing for an equal distribution of data points on either side of each central value. This is fundamental in maintaining essential properties of the signal like its zeroth and first moments. In this context, this corresponds to maintaining the total area under each segment and the mean position of any feature respectively [26]. Moreover, as we move along the window for every time step t, it is clear that when sampling near the edges at $t = 0$ and $t = T$ there will not be any existing data to the left or right. To address this, padding is used for all segments that require it. Specifically, our chosen implementation utilizes that of interpolatory padding, which mirrors the internal data points at the boundaries to generate the padded values [29]. For instance, with the data array $[a, b, c]$ and a window size W of 5, a single padding value is needed on both ends meaning the padded array becomes $[b, a, b, c, b]$ to ensure smooth interpolation at the boundaries.

With the parameters set, the primary part of the algorithm is the computation of the least squares polynomial fits. To carry out the least squares fitting, we have to find the coefficients of a polynomial $p(x)$ of degree n that best approximates the data points in a given window. For a window of size $W = 2M + 1$ centered at a time t, we gather our observations as samples from

$x(t-M)$ to $x(t+M)$. The polynomial for this window is represented as:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, \tag{19}$$

where $\boldsymbol{a} = [a_0, a_1, \ldots, a_n]^T$ is the column vector of polynomial coefficients. If we store the data samples in a column vector $\mathbf{Y}$ and construct the corresponding Vandermonde matrix $\mathbf{X}$ for a window centered at time t, we have:

$$\mathbf{Y} = \begin{bmatrix} y(t-M) \\ y(t-M+1) \\ \vdots \\ y(t+M-1) \\ y(t+M) \end{bmatrix} \quad \text{and} \quad \mathbf{X} = \begin{bmatrix} 1 & x(t-M) & x(t-M)^2 & \ldots & x(t-M)^n \\ 1 & x(t-M+1) & x(t-M+1)^2 & \ldots & x(t-M+1)^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x(t+M-1) & x(t+M-1)^2 & \ldots & x(t+M-1)^n \\ 1 & x(t+M) & x(t+M)^2 & \ldots & x(t+M)^n \end{bmatrix}.$$

With these, the aim now is to minimize the error between the predictions generated by the polynomial model $\mathbf{X}\boldsymbol{a}$ and the actual data measurements $\mathbf{Y}$. Employing the least squares criterion, this corresponds to minimizing the sum of squared differences between the model predictions and data, or mathematically:

$$\min_{\boldsymbol{a}} \|\mathbf{Y} - \mathbf{X}\boldsymbol{a}\|_2^2 . \tag{20}$$

To solve this optimization problem and obtain the polynomial coefficients $\boldsymbol{a}$, we apply the normal equations [30]:

$$\mathbf{X}^T\mathbf{X}\boldsymbol{a} = \mathbf{X}^T\mathbf{Y}. \tag{21}$$

Therefore, for each window we have a closed form solution for $\boldsymbol{a}$ that can be directly computed through:

$$\boldsymbol{a} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}. \tag{22}$$

Having determined the optimal polynomial coefficients $\boldsymbol{a}$ for a window centered at the t, the polynomial $p(x)$ in eq. (19) is now known. The filter output at this time t is simply the value of the polynomial at the centre of the window. Graphically this looks like:

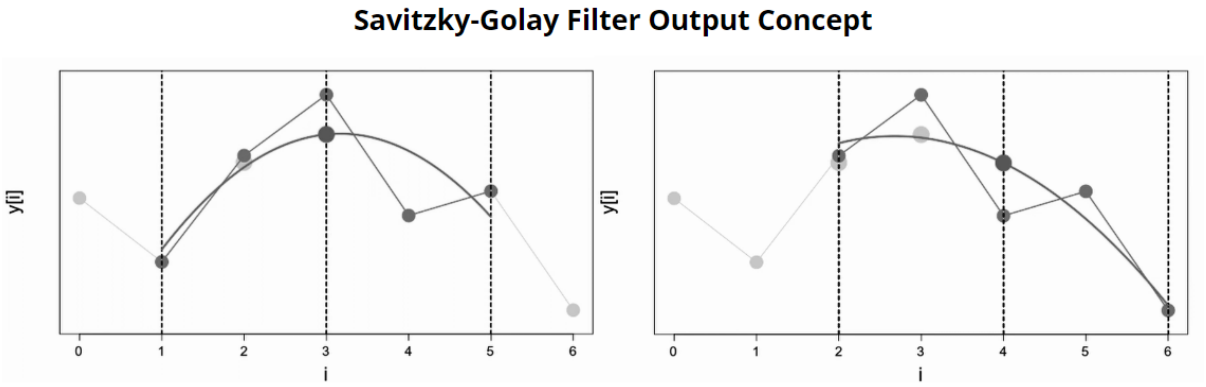


Figure 3: Showcase of filter output ($W = 5$, $n = 2$) following least squares polynomial fit using centre values of each window taken from a simulation by Kiyono Lab [31].

This procedure is repeated for each window to produce a series of smoothed central values which, when combined sequentially, construct the final Savitzky-Golay filtered signal.

3.3 ℓ_1 Trend Filtering (Global)

Introduced by Seung-Jean Kim et al. in 2009 [24], the global method of ℓ_1 trend filtering is a variation of the widely used Hodrick-Prescott filter in the field of trend estimation. The original Hodrick-Prescott filter is based on finding the optimal sequence of new data points $\hat{x}$ that minimizes the weighted sum objective function:

$$\hat{x} = \arg\min_{x \in \mathbb{R}^n} \left(\frac{1}{2} \sum_{t=1}^{n} (y_t - x_t)^2 + \lambda \sum_{t=2}^{n-1} (x_{t-1} - 2x_t + x_{t+1})^2 \right), \tag{23}$$

where the first component is the data fidelity term that ensures the estimated trend x follows the observed data y through computing the residual, and the second one is the smoothness penalty term with a strength parameter λ. The expression $x_{t-1} - 2x_t + x_{t+1}$ is the discrete second difference at time t, which in the realm of trend filtering can detect changes in slope as it is only zero if the three points x_{t-1}, x_t, and x_{t+1} are on a line [24].

Rather than using the sum of squares as the constituent component of the smoothness penalty term, the idea of the ℓ_1 method is to replace this part with the ℓ_1 norm, or the sum of the absolute values of all elements of a vector. The motivation behind this was to address the issue of the Hodrick-Prescott filter often over-smoothing data that contain abrupt changes or shifts, which consequently would lead to certain dynamics being concealed. The result of transitioning to the ℓ_1 term results in trend estimates that are piecewise linear, distinctly exhibiting kinks or knots at points of significant change [24]. These further pieces of information can provide valuable insights into the underlying dynamics of the data, which is something that could have been missed otherwise. With this change, our weighted sum objective function is instead:

$$\hat{x} = \arg\min_{x \in \mathbb{R}^n} \left(\frac{1}{2} \sum_{t=1}^{n} (y_t - x_t)^2 + \lambda \sum_{t=2}^{n-1} |x_{t-1} - 2x_t + x_{t+1}| \right), \tag{24}$$

where the terms are the same as before except that the ℓ_1 norm is employed. In contrast to the local Savitzky-Golay filtering method that requires two parameters, the ℓ_1 trend filter only requires the smoothing penalty strength λ to be specified. Like before, we note that a certain optimal λ exists but that this will always be dependent on the noise corruption present in the data. This is further discussed in Section 3.4.

In solving this optimization problem, the discrete second order difference can be represented as a matrix-vector product $\mathbf{Dx}$ where the matrix $\mathbf{D}$ is defined as:

$$\mathbf{D} = \begin{bmatrix} -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & 2 & -1 \end{bmatrix} \in \mathbb{R}^{(n-2)\times n}.$$

This thus means that our optimization problem can be expressed as:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \left(\frac{1}{2} \|\mathbf{y} - \mathbf{x}\|_2^2 + \lambda \|\mathbf{Dx}\|_1 \right). \tag{25}$$

Compared to the case of Savitzky-Golay filtering, the problem in eq. (25) does not have a closed form solution as it is a convex optimization problem. This means that the use of a numerical optimization technique is required to find the solution, which makes this filtering approach significantly more computationally expensive than that of the former even though there is only one global data frame to consider. To solve this problem, we make use of the open source Python package `cvxpy` [32] for convex optimization problems. Specifically, we first try using their embedded conic solver (ECOS) [33] implementation, and if that fails we default to the splitting conic solver (SCS) [34] approach, noting that the mathematics of these techniques are outside the scope of this study.

```python
import cvxpy as cp

def l1_trend_filter(y, lamb=0):
    n = len(y)
    x = cp.Variable(n)
    # Construct 2nd order difference matrix D2
    D = np.diff(np.eye(n), axis=0)
    D2 = np.diff(D, axis=0)
    # Define the objective
    objective = cp.Minimize(0.5 * cp.norm(x - y, 2)**2 + lamb * cp.norm(D2 @ x, 1))
    # Construct the problem
    problem = cp.Problem(objective)
    # Apply ECOS solver
    try:
        problem.solve()
    except cp.SolverError:
        # Apply SCS solver if ECOS fails
        try:
            problem.solve(solver=cp.SCS)
        # No solution found
        except cp.SolverError:
            return None
    # Return the solution
    return x.value
```

Figure 4: Implementation of ℓ_1 trend filter in Python showing solver order with ECOS as default and SCS as backup.

3.4 Pareto L-Curve Criterion - Optimal Hyperparameter

The filtering methods discussed so far necessitate the selection of specific hyperparameters, this being the window size W for the Savitzky-Golay filter and the smoothing penalty strength λ for the ℓ_1 trend filter. In the context of our study, these are termed hyperparameters because their values are not determined by the filtering method itself but are instead chosen to optimize the filter's performance on a specific task.

Choosing appropriate hyperparameter values is challenging because we cannot assume knowledge about the noise level in the data or the exact form of the underlying dynamics. If we had such information, we could tailor our filtering approach to specifically capture the true dynamics. However, in the absence of this knowledge, we are left to balance the trade off between over-smoothing the data, which might obliterate crucial dynamics, and under-smoothing, which might retain excessive noise.

There are various data driven methods aimed at making this optimal hyperparameter selection. One of these is the method of cross-validation (CV) [35]. In a k-fold CV approach, the dataset is randomly partitioned into k equal sized subsamples of which $k-1$ are used for training and the remaining for validation, with the process repeated k times. For each parameter selection investigated, we thus get an average validation error over the k trials with the goal being to pick the one with the lowest error. However, a challenge specific to our context is setting a meaningful validation target. Since we cannot assume the true shape of the data signal representing the dynamics or the noise structure imposed, the validation data often defaults to the initial noisy measurements. This can complicate the determination of an optimal hyperparameter as the selection process is guided to select a parameter that resembles the original noisy data, noting that this is not what we want. Therefore, we make use of the Pareto L-curve criterion [25] instead which has been found to work well in this setting [10].

The premise of using a Pareto criterion is to find a parameter that balances the acts of feature preservation and smoothing. Mathematically, this corresponds to identifying the parameter that gives the most optimal trade off between the residual (residual norm) and regularization impact (solution norm). When these measures are plotted over a range of hyperparameter values, we tend to get an L-curve shape where the point of maximum curvature corresponds to the optimal trade off we are after. For clarity, this is often graphed on a log-log plot. If we consider a noise corrupted dataset $\mathbf{x}'$ and apply any filter to produce the denoised dataset $\mathbf{x} = F(\mathbf{x}', \alpha)$, where α is the currently selected hyperparameter, we calculate the residual norm and solution norm as follows:

$$\text{solution norm}(\alpha) = \|\mathbf{x}(\alpha)\|_2, \tag{26}$$

$$\text{residual norm}(\alpha) = \|\mathbf{x}' - \mathbf{x}(\alpha)\|_2. \tag{27}$$

When these values are plotted on their respective axis for the parameter value α, the corresponding curvature at that particular point can be found using the second derivative:

$$\kappa(\alpha) = \frac{d^2(\text{solution norm}(\alpha))}{d(\text{residual norm}(\alpha))^2}. \tag{28}$$

The optimal value of α is the one that maximizes the curvature over all values tested:

$$\alpha_{\text{optimal}} = \arg\max_{\alpha} |\kappa(\alpha)|. \tag{29}$$

To visualize the result of selecting the most optimal hyperparameter through this approach, we consider a test function composed of multiple sinusoids with varying frequencies and amplitudes of the form $f(x) = \sin(x) - \frac{1}{2}\sin(3x) + \frac{1}{3}\sin(5x)$. Generating 500 equally spaced data points between $x \in [0, 4\pi]$ and introducing variable levels of Gaussian noise, we showcase the performance of both the local Savitzky-Golay filter and global ℓ_1 trend filter using Pareto tuning below:

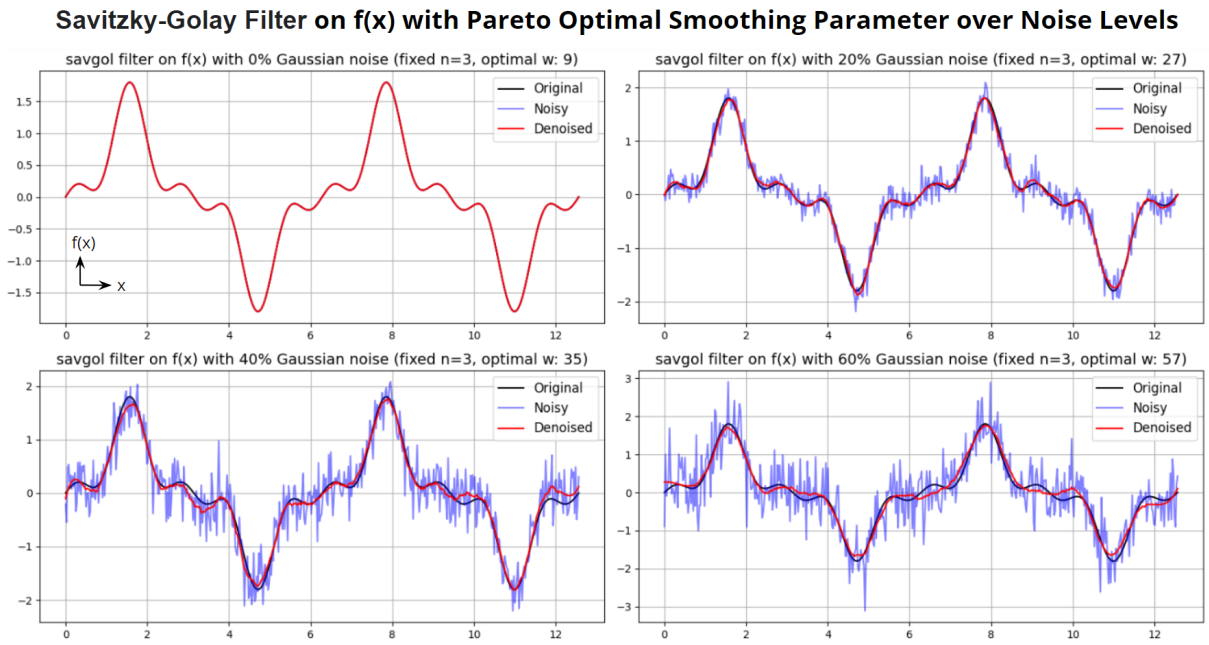


Figure 5: Pareto tuned Savitzky-Golay filter (for evenly spaced W values in the range 5 to 101 in increments of 2) fittings for the function $f(x)$ under Gaussian noise additions of 0, 20, 40, and 60 percent. A large-scale image is included in the Appendix for improved clarity.

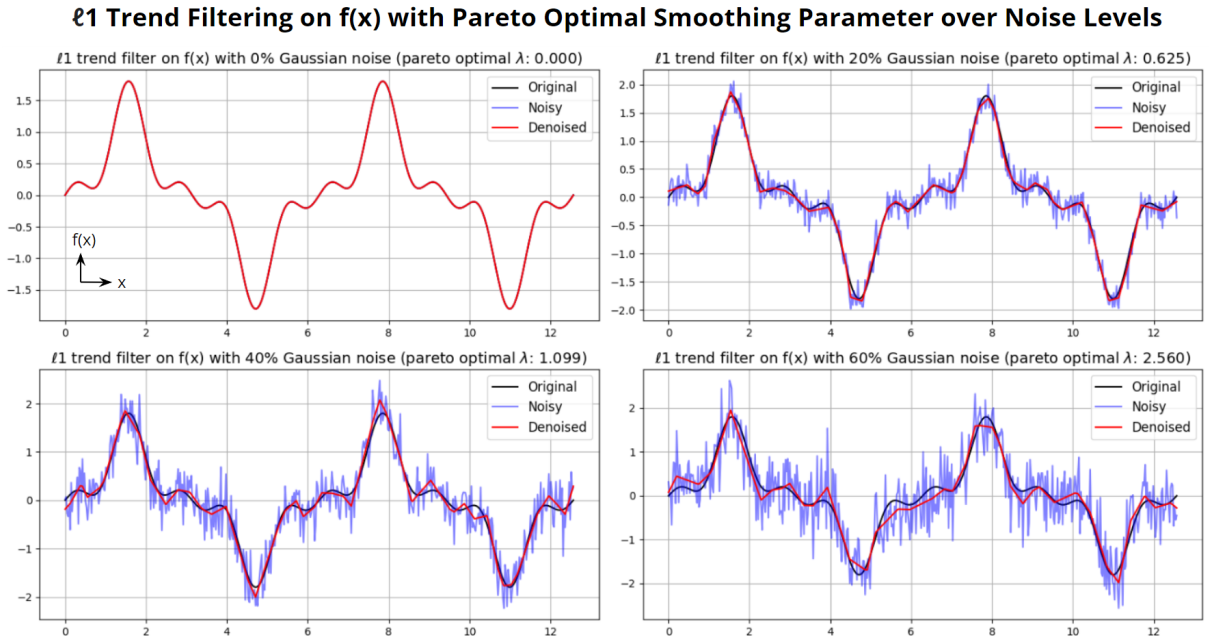


Figure 6: Pareto tuned ℓ_1 filter (for 50 evenly spaced λ values over the magnitude range 10^{-4} to 10^2) fittings for the function $f(x)$ under Gaussian noise additions of 0, 20, 40, and 60 percent. A large-scale image is included in the Appendix for improved clarity.

It is clear that this selection method for obtaining a suitable filtering parameter is powerful, with both filters yielding a denoised signal that bears significant resemblance to the original function even at high noise levels. We note, however, that performance does indeed begin to fall off as the noise percentage is increasing, particularly so at the 60% level. With the ℓ_1 trend filter at this noise level, areas of significant feature shifts like at $x \in (2, 4)$, $x \in (5.5, 7)$, and $x \in (8.5, 10)$ see sharp turns in the filtered signal whereas the true function is much smoother. This is in line with the kinks that are expected from having adopted the ℓ_1 based penalty term. On the other hand, although the Savitzky-Golay filter is able to preserve the features in these regions more closely, the signal itself is more jagged at these points upon closer inspection. Both of these problems result in deviation from the true form, motivating the idea of using a further filter to smooth out these areas.

To also illustrate the concept of the L-curve and the point of maximum curvature, the corresponding log-log plot for the selection of the optimal ℓ_1 λ in the 40% noise case is shown below:

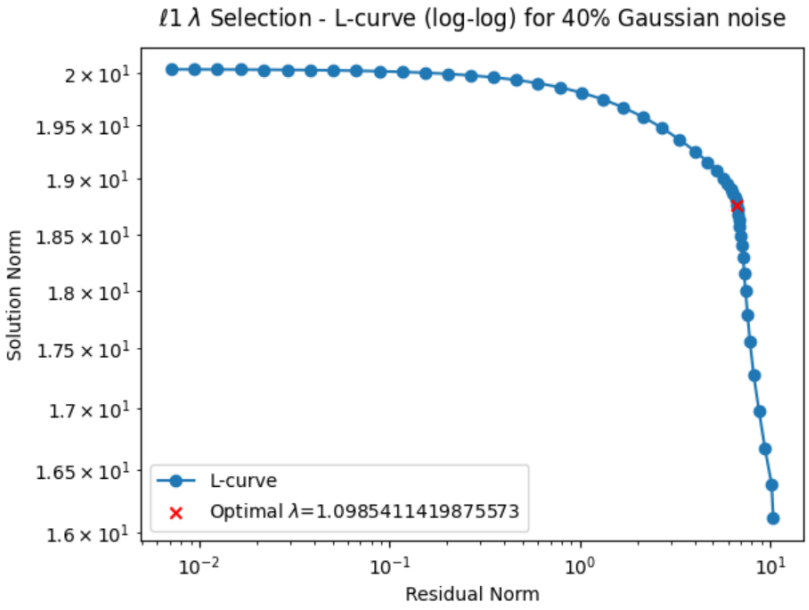


Figure 7: L-curve (log-log form) corresponding to the ℓ_1 λ selection in the 40% noise case.

3.5 Proposed Two-Stage Filter Designs

To achieve more precise identification of the genuine dynamics inherent within a signal corrupted by a high-level noise, we propose the two-stage filtering approach of stacking the two filtering methods already discussed. The inspiration for this design emanates from the foundational architecture of convolutional neural networks (CNNs). At their core, CNNs are structured to exploit multiple layers or components, each responsible for progressively refining and abstracting the information. This layered and iterative process ensures that each subsequent stage builds upon the refined outputs of its preceding stage, ultimately leading to a more holistic and informed representation of the original input.

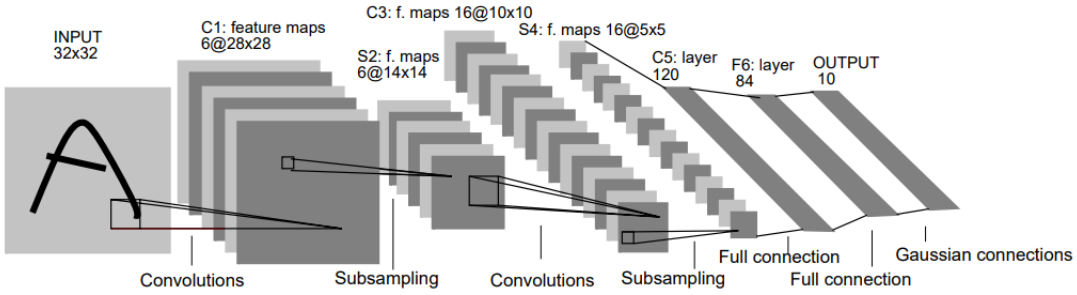


Figure 8: LeNet-5 CNN architecture for hand-written digit recognition by LeCun et al. [36].

However, our motivation to investigate a two-stage approach is not solely based on the premise of CNNs. As observed in the previous section of our study, even under an optimal regularization hyperparameter selection the use of both filters, although particularly clear with the ℓ_1 approach, resulted in undesirable jaggedness in the reconstructed signal at higher noise levels. Inconsistencies like these compromise the reliability of the denoised output, making it difficult to discern genuine features from artifacts introduced by the noise. Through leveraging a two-stage filtering process, we aim to integrate the strengths of both the local and global filtering methods by performing them in sequence. The initial stage of the filter effectively handles the broader trends within the data, ensuring that significant features and dynamics are preserved. Once this foundational restoration is achieved, the second stage refines this output, focusing on localized discrepancies and anomalies to address the previously observed jagged inconsistencies.

Furthermore, in our investigation we remain methodologically agnostic about the order of filtering stages. Recognizing the potential variances in outcomes dependent on the sequence, we examine both directions: global filter followed by local and vice versa. By exploring both sequences, we aim to identify the most effective combination for signal reconstruction. Additionally, we recognize that the optimal order might be influenced by the specific characteristics of the system under study, motivating the study of both combinations as well.

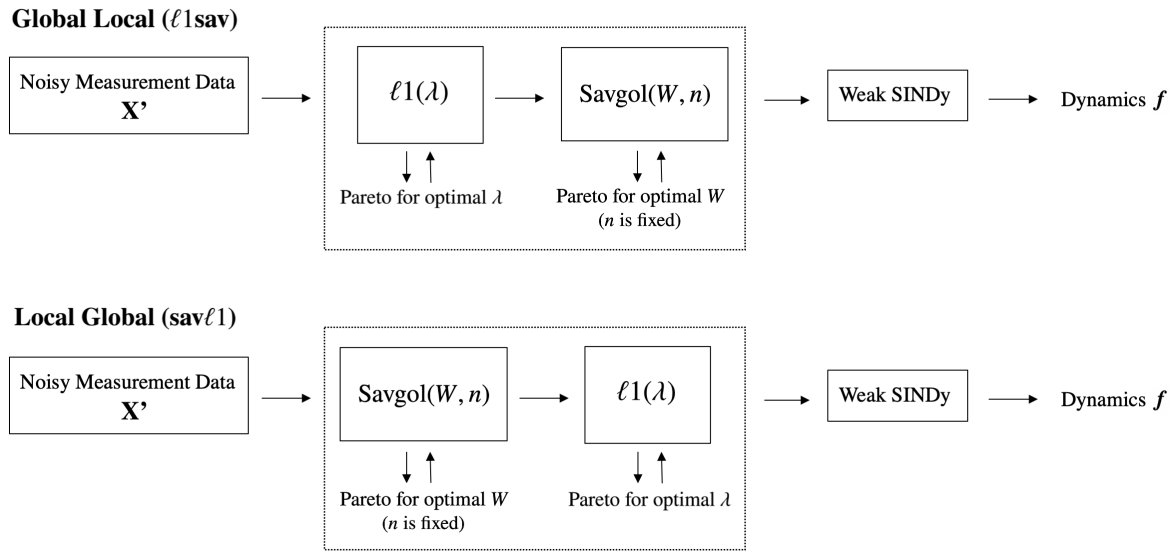


Figure 9: Two-stage filter combinations investigated for identification of nonlinear dynamics in the presence of noise.

To gain an initial understanding of how a two-stage approach like this performs especially in terms of mitigating the jaggedness observed with a single filter, we revisit the previously defined $f(x) = \sin(x) - \frac{1}{2}\sin(3x) + \frac{1}{3}\sin(5x)$ before moving onto governing dynamics. Our focus is particularly on the instance where 60% Gaussian noise had been introduced and observable deterioration had started occurring. As an illustrative example, we use the exact same setup for data generation as before but instead of only considering the singular filters, we also apply the corresponding secondary filters in line with the design above to investigate their effectiveness.

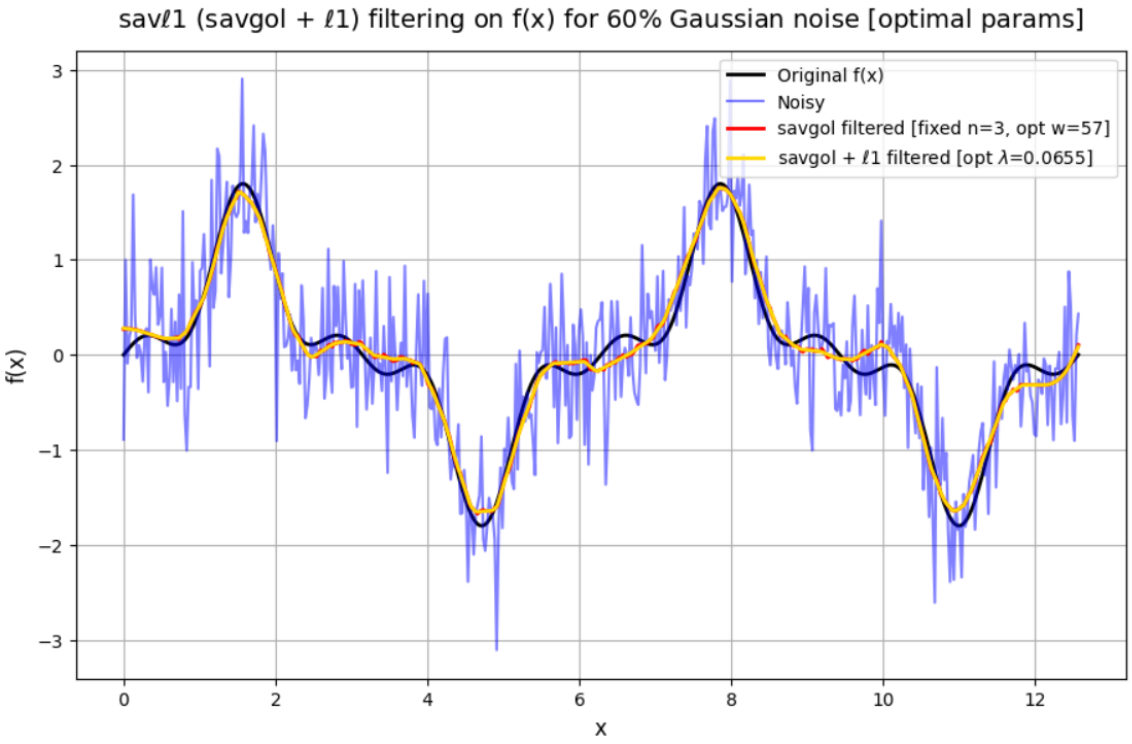


Figure 10: Savitzky-Golay and savℓ_1 filtered signals of $f(x)$ under 60% Gaussian noise.

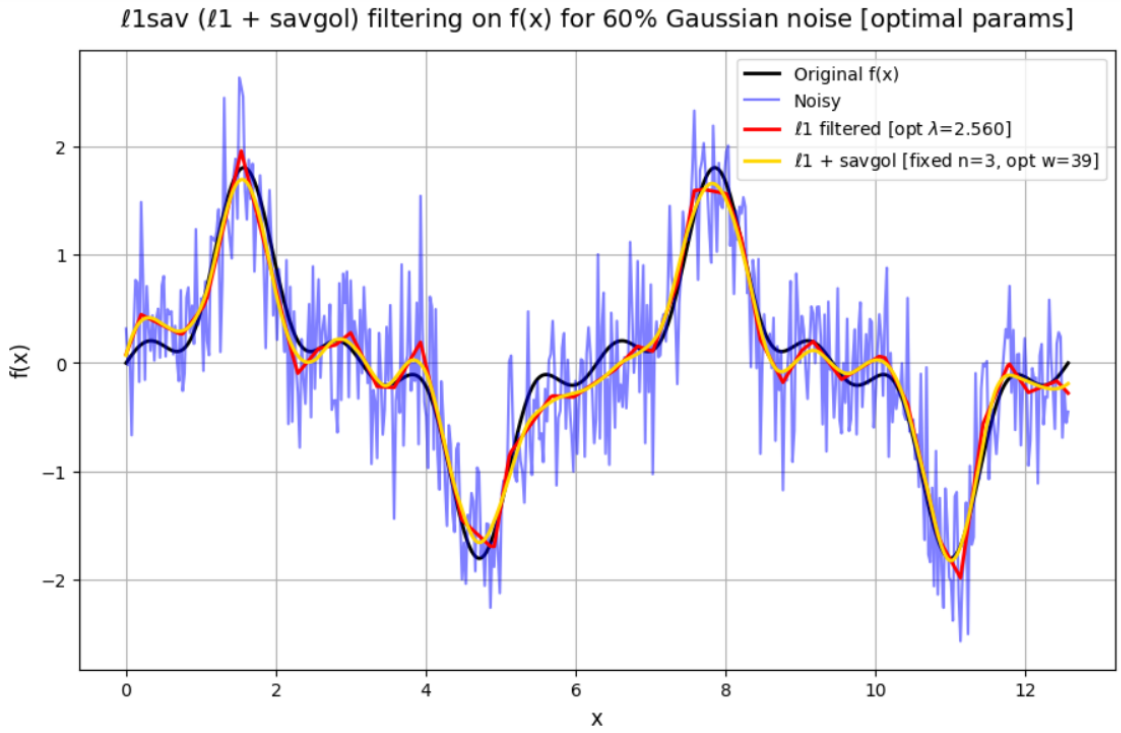


Figure 11: ℓ_1 trend filter and $\ell_1$sav filtered signals of $f(x)$ under 60% Gaussian noise.

In the figures presented, the result of applying a secondary filter as well is clearly visible. In the case of the savℓ_1 design, although the two-stage filtered signal follows the initial Savitzky-Golay output extremely closely, the smaller-scale jaggedness in the areas previously highlighted have been eliminated. Even though the qualitative change is minor, we note that a difference still exists meaning that the design remains of interest to investigate further. With the ℓ1sav approach, on the other hand, the effect is much more noticeable. It effectively manages to smooth out the kinks generated by the standalone ℓ_1 filter, making the denoised signal much more in line with the shape of the true function. Additionally, we note that by having addressed these sharp turns, the two-stage denoised output matches the major peaks and troughs of the true signal significantly better. It is important to highlight that these effects may be more pronounced when considering a more complex signal than the one created by $f(x)$. With our goal being to ultimately tell whether this improves the successful identification of the nonlinear governing dynamics responsible for generating these signals, it is difficult to say at this stage already whether this will help and is the focus of the next two chapters.

4 Experimental Methodology

4.1 Gaussian Noise Addition

Throughout the course of this investigation, we have consistently employed Gaussian noise to simulate real-world uncertainties and imperfections inherent with data collection. While various types of noise such as impulsive and uniform exist [37], Gaussian noise is predominantly favored and tends to be the norm in studies like ours. One reason for this is the ubiquity of the Gaussian distribution in natural processes given its bell-shaped probability distribution [38]. Additionally, the Central Limit Theorem suggests that the aggregation of numerous small independent random corruption processes during data acquisition tends to converge to a Gaussian distribution [39]. This implies that random noise, especially when stemming from a myriad of sources, is most likely of Gaussian form. As we continue to use this form moving forwards, we outline the mathematical formulation below for completeness.

Given our original dataset x_i where i represents the specific data point, the introduction of Gaussian noise can be expressed using the equation:

$$x_i' = x_i + \epsilon_i. \tag{30}$$

In this context, ϵ_i represents a random variable derived from a Gaussian or normal distribution, formally represented as:

$$\epsilon_i \sim \mathcal{N}(\mu, \sigma^2), \tag{31}$$

where μ is the mean and σ the standard deviation. Here the mean μ is zeroed to ensure that the noise distribution remains centered around the original data, with the standard deviation σ being our point of noise introduction. In order to be able to set an intensity level to this noise, we first compute the Root Mean Square Error (RMSE) of the dataset relative to the zero vector. This is to capture a measure of the relative magnitude of the values present in the data, meaning that we can thus introduce a strength level that remains proportional to the specific data in question.

If we do this and define a percentage as our measure of added intensity, we have the following formulation:

$$\sigma = \left(\frac{\text{percentage}}{100}\right) \times \text{RMSE}(x_i, 0) = \left(\frac{\text{percentage}}{100}\right) \times \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}. \tag{32}$$

Therefore, our intensity-specified Gaussian noise ϵ_i for a dataset x_i takes the form:

$$\epsilon_i \sim \mathcal{N}\left(0, \left(\left(\frac{\text{percentage}}{100}\right) \times \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right)^2\right). \tag{33}$$

4.2 Primary Libraries Used

To handle all of the SINDy related computations, the open source Python package `PySINDy` [40] was utilized. This library has, and continuous to be, under constant development to implement the various new strategies that continue to be studied in relation to SINDy. Crucially, the module `WeakPDELibrary` allows for approaching the problem under study (both ODE/PDE based) using the weak formulation, which is what we require for our baseline algorithm. Furthermore, it also contains the necessary regularization optimizers needed depending on whether we are in an ODE or PDE setting.

In regards to the filtering methods, it has already been shown how `cvxpy` is utilized in solving the optimization problem for the ℓ_1 filter, but for the Savizty-Golay method many complete implementations exist where we list our chosen one below. In both cases, the procedure to find the Pareto optimal filter hyperparameter has to be implemented separately.

Filtering Method	**Python Library**
Savitzky-Golay	`scipy.signal.savgol_filter` [29]
ℓ_1 trend filter	`cvxpy` [32]

Table 1: Filtering methods and their corresponding libraries.

4.3 Error Measurement Metrics

Determining the underlying dynamics of a system based on observed data is inherently a complex task, and this branches into the objective of finding an appropriate error metric as well. As we are dealing with results in the form of equations, we have to consider both the terms themselves as well as their coefficients. This however creates various intricacies. In high noise settings, it is highly likely that the discovered dynamics or equation contains additional terms that are not present in the targeted, or true, dynamics. Conversely, the discovered equation might miss out on essential terms that are in fact present in the true dynamics. Quantifying the impact of these alterations is challenging because terms are structurally different and will have varying influences on the dynamics. Even in the simplest case where all the true terms are correctly identified and only coefficients vary, it is still not trivial to produce a single score to measure how correct the discovered dynamics are due to this reason. Given these challenges, we employ a suite of metrics each targeting a different aspect of the discovery process to allow for a more holistic assessment.

We define these over N total iterations as follows:

1. **Success Rate:** Percentage of time all terms in the discovered model are fully correct.

   Let D denote a set containing the discovered terms and T represent the set of true terms. The success rate S is then given by:

   $$S = \frac{\text{count } (D = T)}{N} \times 100\%, \tag{34}$$

   where a successful iteration $D = T$ implies that D matches T exactly without any additional terms. This measure is irrespective of coefficient magnitude as long as the coefficient of any found term is above the defined threshold for keeping terms. This threshold θ is set to 5×10^{-4} within our experiments.

2. **Average Total Coefficient Error:** The total deviation between the coefficients of the discovered dynamics and the true dynamics averaged over all iterations.

   Given coefficients vectors $\mathbf{c}_D$ for the discovered dynamics and $\mathbf{c}_T$ for the true dynamics, the average total coefficient error E_C over all iterations N is:

$$E_c = \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{m=1}^{M} |\mathbf{c}_{T_m} - \mathbf{c}_{D_m}| \right), \tag{35}$$

   where M is the maximum number of terms in either D or T across all iterations. For any extra term in D that does not exist in T, the absolute value of its coefficient $|\mathbf{c}_{D_m}|$ is added to the error. Moreover, for any term missing in D that exists in T, the absolute value of the corresponding true coefficient $|\mathbf{c}_{T_m}|$ is added to the error.

3. **Average Number of Wrong Terms:** The count of the number of wrong terms in the discovered dynamics when compared to the true dynamics averaged over all iterations.

   Given the sets D and T, we can define W as the set difference between them to capture any additional terms. Then, the average number of wrong terms E_w is:

$$E_w = \frac{1}{N} \sum_{j=1}^{N} |W_j|, \tag{36}$$

   where W_j refers to the set of wrong terms in the j^{th} iteration.

Having defined all of these, it is important to re-iterate that while these metrics provide valuable insights into the accuracy of the discovery process, none of them is perfect and their interpretation must be contextual. For example, a low coefficient error in a less influential term might be less critical than a slightly higher error in a dominant term making further qualitative analysis also important.

4.4 Main Simulation Loop and Remarks

In order to test the effectiveness of the standalone filtering methods investigated as well as the proposed two-stage formulations, we devise the following test loop for a set dynamical system under study, being either the Lorenz ODE System or the Viscous Burgers' PDE:

Algorithm 2 Testing System Dynamics Recovery with Noise and Filtering

procedure SYSTEMTEST
    Define list of Gaussian noise percentages $\mathcal{P}$
    Define threshold θ, domain number K, library functions $\mathcal{F}$, regularizer $R(\hat{\boldsymbol{\Xi}})$
    Define Savitzky-Golay polynomial degree n
    Load/generate system data
    Initialize true system coefficients c_T
    for noise percentage p in $\mathcal{P}$ **do**
        Initialize error metrics
        for iteration $i = 1$ **to** N **do**
            Introduce Gaussian noise percentage p to data
            Pareto optimal parameter selection for ℓ_1 λ and Saviztky-Golay W
            Apply ℓ_1 and Savitzky-Golay filters
            Pareto optimal parameter selection for two-stage filters savℓ1 and ℓ1sav
            Apply savℓ1 and ℓ1sav filters
            for each filtered data **do**
                Fit system dynamics using library and regularization optimizer
                Extract active terms and coefficients from the discovered model
                Compute error metrics
            end for
        end for
        Output error metrics averaged over all iterations N
    end for
end procedure

Since the addition of Gaussian noise is a random process with values being drawn from a defined distribution, corruption by a fixed strength level will not produce identical noisy datasets over multiple runs. This inherent randomness means that even for closely set noise percentages, there's no guarantee that a lower percentage will always introduce less corruption than a slightly higher one in a single instance. To address this variability and ensure more reliable and consistent results, we use an averaging approach. By running each noise level over a large number of iterations, $N = 500$ in our case, we allow the results to converge to a more representative average by smoothing out any random fluctuations. The ℓ_1 filtering process, in contrast to the

Savitzky-Golay method with its closed-form solution, necessitates tackling a computationally demanding convex optimization problem as mentioned previously. This complexity is particularly heightened during the Pareto optimal selection, wherein a diverse set of parameter values undergo examination. Given this significant computational load, further amplified by the high iteration count, we harnessed the capabilities of the university's high-performance supercomputer BlueBEAR. Through parallelization where each noise level was tested concurrently, we were able to reduce the computational overhead significantly.

With one of our experimental studies containing multiple measured variables, x, y, z for the Lorenz model, it is of high importance to highlight the need to recognize and maintain their intrinsic scales. When applying a transformation such as filtering to the dataset, combining measurements of diverse magnitudes can influence results. These differences may be misconstrued as noise during the Pareto selection of the optimal filtering parameter, consequently affecting the filtering strength and therefore the final outcome. As a result, when processing multivariate data, we handle each variable individually in the filtering process.

As the goal of this study lay in the efficacy of different filter design to discern the governing dynamics of a system, we assume a well constructed library of candidate terms. This means that in both our studies, the true terms that are in fact present in the target model is included as options that we perform the regression on. Furthermore, in order to devise a fair test in the most optimal of conditions, we also utilize parameters values for the weak SINDy algorithm that have been shown to produce robust results for the general case with no filtering. This is necessary to attribute any observed variations in performance to the filtering technique under examination. From the PySINDy documentation [41], we select (having defined the threshold $\theta = 5 \times 10^{-4}$):
Lorenz ODE: Default STLSQ Regularizer with $\eta = 0.1, \lambda = 0.05$ and $K = 100$ different domains (setting Savitzky-Golay polynomial degree to $n = 2$).
Viscous Burgers' PDE: SR3 Regularizer with $R(\mathbf{W}) = \ell_0(W)$, $\nu = 1, \lambda = 5 \times 10^{-3}$ and $K = 1000$ different domains (setting Savitzky-Golay polynomial degree to $n = 3$).

5 Results on Canonical Systems

5.1 Lorenz System (ODEs)

First studied by Edward Lorenz in 1963, the Lorenz system compromises a set of three ordinary differential equations that model the atmospheric convection of fluid flow [42]. Specifically, it represents a two-dimensional fluid layer heated from below and cooled from above, which is a scenario known as Rayleigh-Bénard convection [43]. The equations take the form with nonlinearity in $\dot{y}$ and $\dot{z}$:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{37}$$

where σ is the Prandtl number, ρ the Rayleigh number, and β a geometric factor. This system is particularly well known for its stability patterns and chaotic solutions, where only slight changes in initial conditions and parameters can result in significant variations in the solution [44]. For the classical chaotic behaviour observed by Lorenz that we also adopt, the parameters are set to $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$ to give:

$$\begin{aligned} \frac{dx}{dt} &= 10(y - x), \\ \frac{dy}{dt} &= x(28 - z) - y, \\ \frac{dz}{dt} &= xy - \frac{8z}{3}. \end{aligned} \tag{38}$$

In generating the training data, to ensure that we are not restricted by a low data limit [6] or too large of a sampling rate [13], we solve the system forwards in time using a step size of $dt = 0.001$ between $t = 0$ and $t = 8$ with initial conditions $x = -8, y = 8, z = 27$. This can be done using SciPy's `solve_ivp` function within their `integrate` library [45]. This yields 8000 different measurement points for each variable and can be visualized using a 3-dimensional plot as follows:

```python
from scipy.integrate import solve_ivp

# System definition
def lorenz(t, X, sigma=10, beta=8/3, rho=28):
    x, y, z = X
    dx = sigma * (y - x)
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return [dx, dy, dz]

# Initialization
dt = 0.001
integrator_keywords = {}
feature_names = ['x', 'y', 'z']
t_train = np.arange(0, 8, dt)
x0_train = [-8, 8, 27]
t_train_span = (t_train[0], t_train[-1])
# Solve forwards in time
x_train_original = solve_ivp(lorenz, t_train_span, x0_train, t_eval=t_train, **integrator_keywords).y.T
```

Figure 12: Training data generation code in Python.

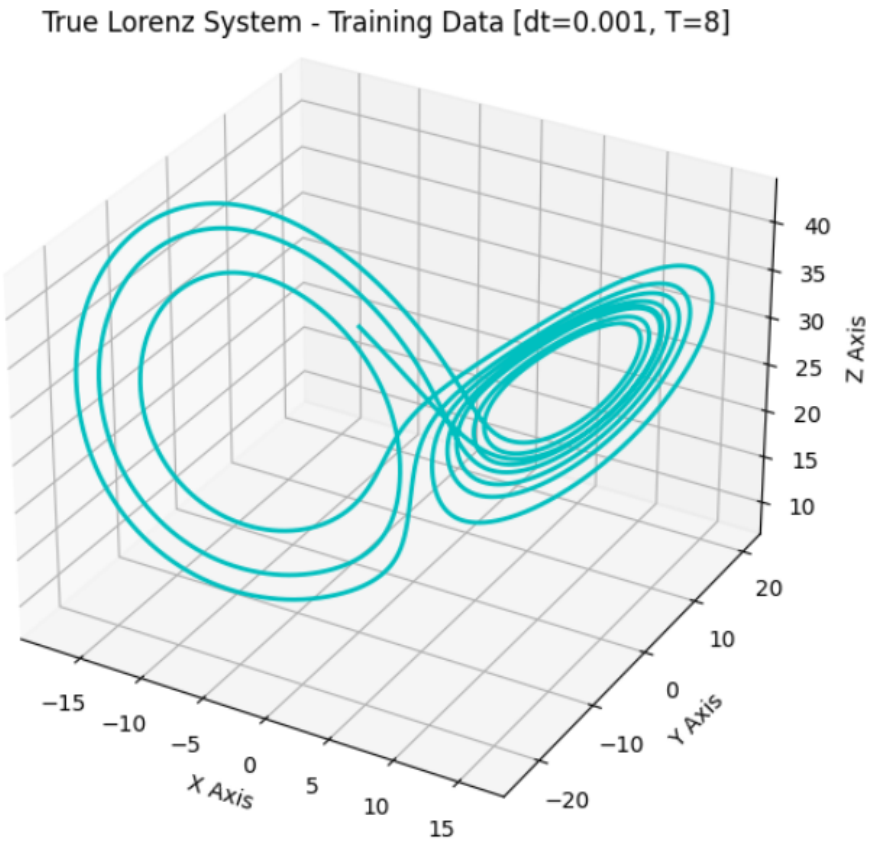


Figure 13: Corresponding plot representing the true Lorenz dynamics we aim to reconstruct.

Applying the various filtering design as a pre-processing step to the weak formulation in order to reconstruct the defined Lorenz dynamics over a Gaussian noise corruption range of 0-70%, we obtain over 500 iterations:

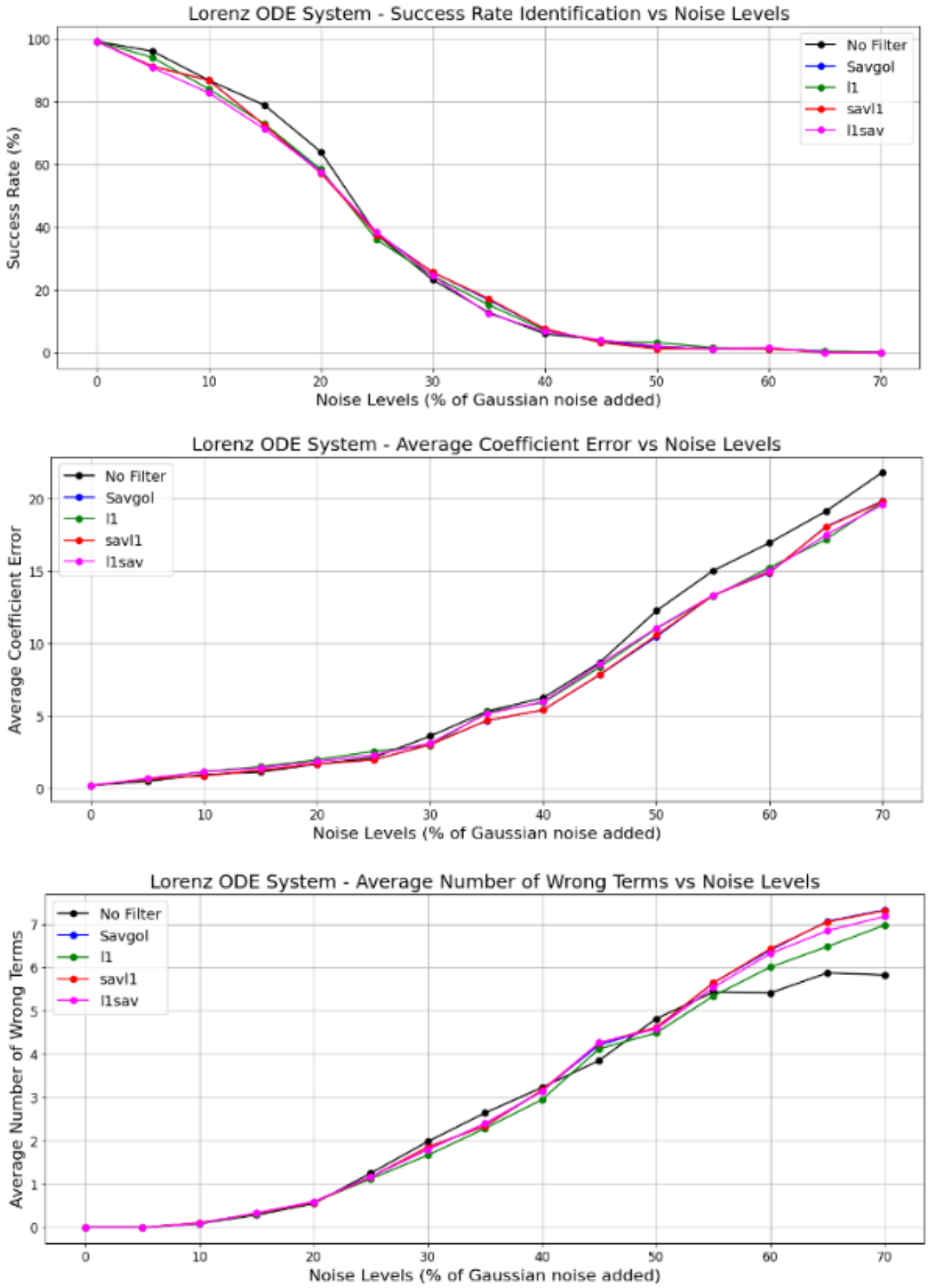


Figure 14: Composite results over all metrics tested with scores of no filtering as baseline.

At initial inspection of the successful identification rate, no major variation is seen other than filtering actually harming this metric for Gaussian noise introduction below the 20% strength. This can be contributed to the issue of over-smoothing where essential features can be classified as noise in the Pareto selection at these lower values. It is important to note, however, that this metric does not account for coefficient error which, even while the success rate is not significantly impacted, sees visible improvement by filtering above the 25% mark. In terms of incorrect number of terms for wrong identification, clear deviation only starts to occur when successful identification becomes near impossible. In this realm, filtering again harms the metric more although we highlight that all forms here are ultimately wrong.

To further investigate the improvement seen in coefficient errors, we extract 10 runs of correct identification by all filtering designs at the 30% Gaussian noise level and average them to yield:

Target Dynamics

$$\dot{x} = -10.000x + 10.000y$$
$$\dot{y} = 28.000x - 1.000y - 1.000xz$$
$$\dot{z} = -2.667z + 1.000xy$$

No Filter $E_c = 3.193$

$$\dot{x} = -10.206x + 10.117y$$
$$\dot{y} = 25.935x - 0.452y - 0.943xz$$
$$\dot{z} = -2.676z + 1.010xy$$

Savitzky-Golay $E_c = 2.028$

$$\dot{x} = -10.182x + 10.090y$$
$$\dot{y} = 26.622x - 0.680y - 0.962xz$$
$$\dot{z} = -2.658z + 1.012xy$$

$l1$ Trend Filter $E_c = 2.533$

$$\dot{x} = -10.205x + 10.116y$$
$$\dot{y} = 26.262x - 0.592y - 0.949xz$$
$$\dot{z} = -2.669z + 1.012xy$$

sav$l1$ Double $E_c = 2.017$

$$\dot{x} = -10.181x + 10.089y$$
$$\dot{y} = 26.628x - 0.683y - 0.962xz$$
$$\dot{z} = -2.658z + 1.012xy$$

$l1$sav Double $E_c = 2.119$

$$\dot{x} = -10.189x + 10.099y$$
$$\dot{y} = 26.572x - 0.654y - 0.961xz$$
$$\dot{z} = -2.660z + 1.012xy$$

Table 2: Average dynamics identified by all methods for 5 correct identification runs where E_c is the total coefficient error of the corresponding model (ignores any measures from wrong identifications).

From these extractions, a clear pattern is visible. Although we have found the success rate values to not vary significantly amongst the techniques in the Lorenz ODE domain, for now correct identifications there is a visual improvement in the total coefficient errors. Both singular methods outperform the no filter baseline, with superior performance seen from the Savitzky-Golay filter. We highlight that each of the double filters both manage to improve the coefficient errors of their respective first components, and that this is likely attributed to the removal of jaggedness as illustrated in Section 3.5. Recognizing the different impact of the terms and that this is a chaotic system, we simulate these systems forwards in time with the same initial conditions:

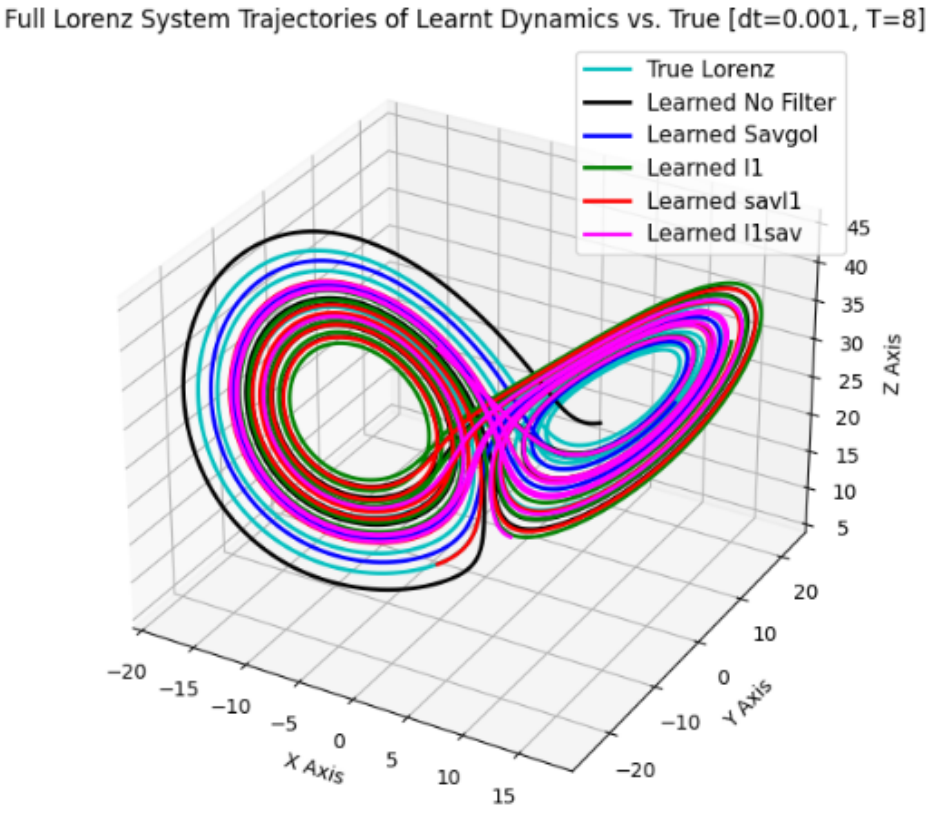


Figure 15: Complete Lorenz trajectories for full simulation period.

Despite there being a minor variation in coefficients, the overall dynamics in terms of shape of the famous attractor is still captured. Deviations can still be seen, most evidently in the outermost curve of the no filter model that is not observed elsewhere. To inspect the results on a closer scale, we plot the last 500 simulated data points for generated by each model with increasing final times T:

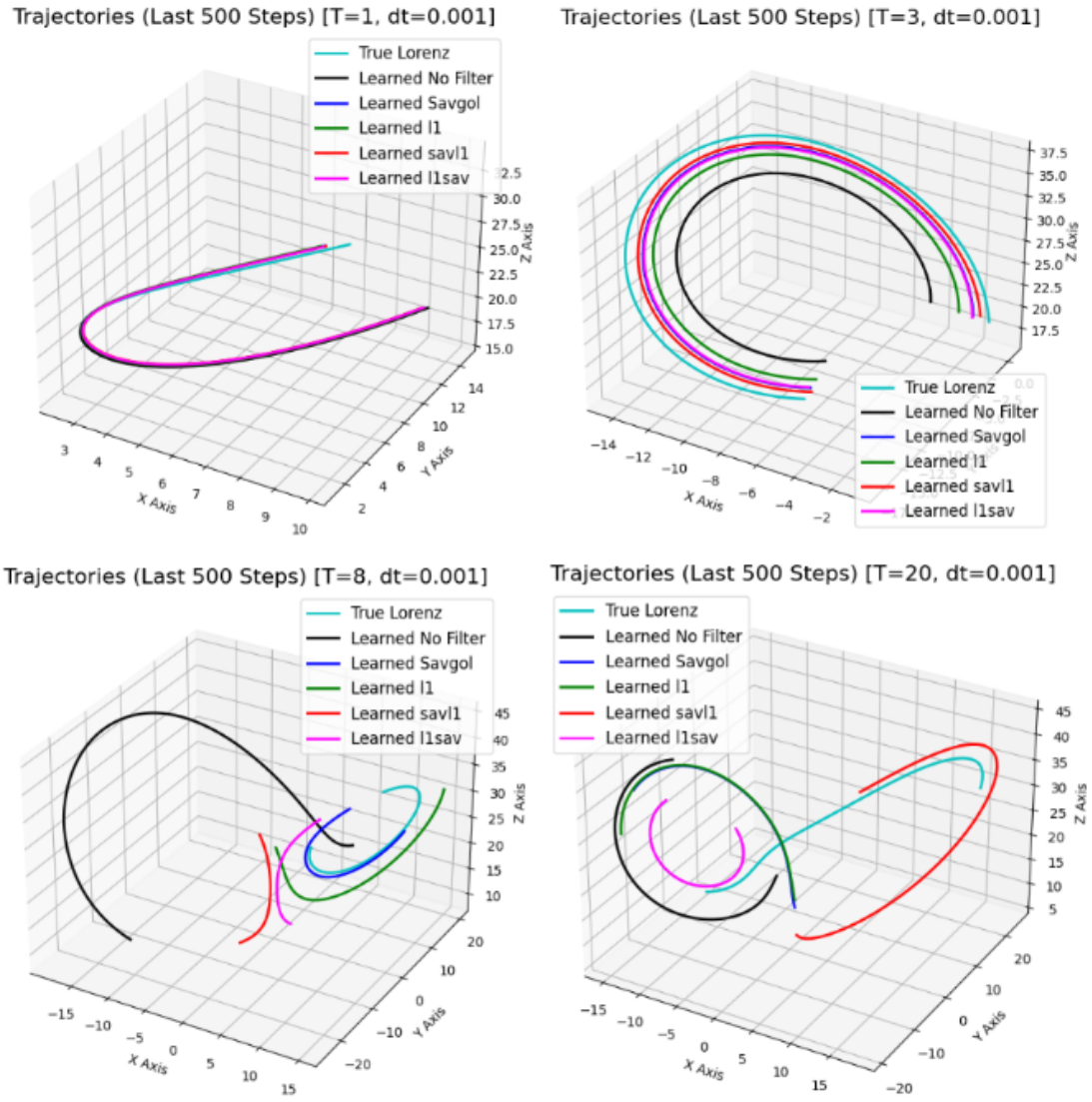


Figure 16: Higher magnification through visualizing last 500 simulated values for varying T.

At this smaller scale, the trajectories representing the behaviour of the dynamics deviate from the true system significantly as the simulation period is extended. In particular, we note that even for a simulation up to $T = 3$, the shape is perfectly maintained by all models. The one without filtering, however, has shifted qualitatively more than the rest. As we move up to $T = 8$, although little resemblance to the true trajectory can be made out from the models, all filtered signals are closer to the correct vicinity. At $T = 20$, no model is close to the correct dynamics. Having already shown the advantage of filtering for the overall trajectory, this further highlights its usefulness for shorter timescale simulations with no major distinctions between the designs except for minor positional improvements on those including the Savitzky-Golay component.

5.2 Viscous Burgers' Equation (PDE)

Burgers' equation is a fundamental nonlinear partial differential equation that captures the interplay of advection and diffusion processes in areas like fluid mechanics, nonlinear acoustics, and gas dynamics. Initially formulated by Harry Bateman in 1915 [46] and further studied by Jan Burgers in 1948 [47], it serves as a model to understand complex turbulent flows and shock phenomena [49]. Presented in its general form noting nonlinearity in the advection term:

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}, \tag{39}$$

where $u(x,t)$ is the velocity field of the fluid and ν the kinematic viscosity quantifying the magnitude of viscous effects in the system. Keeping in line with prior studies on using SINDy to discover this dynamical system [16, 22], we adopt the same training dataset [49] where $\nu = 0.1$. Therefore, the system we aim to reconstruct is:

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} + 0.1\frac{\partial^2 u}{\partial x^2}. \tag{40}$$

Loading the already compiled data file and defining dt and dx as the respective size of adjacent elements in data arrays `t` and `x`, the training data solution $u(x,t) \in \mathbb{R}^{256\times101}$ looks like:

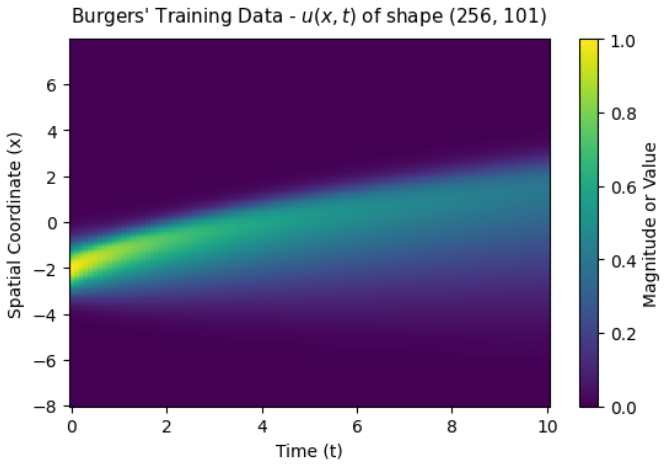


Figure 17: Heatmap visualizing viscous Burger's PDE training data.

Running the same simulation loop defined in Section 4.4 using this training dataset under Gaussian noise addition in the strength range of 0-100%, we obtain over 500 iterations:

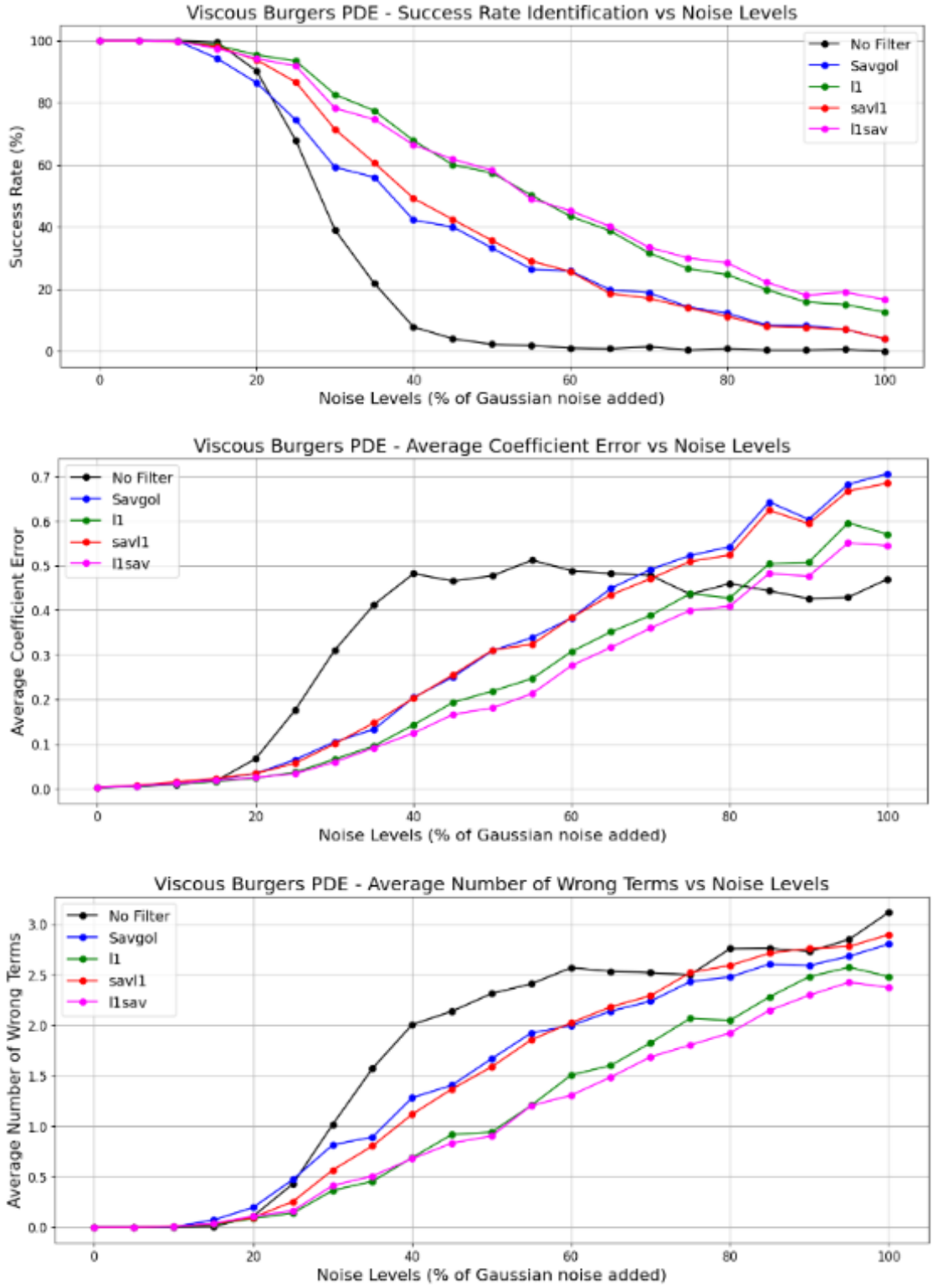


Figure 18: Composite results over all metrics tested with scores of no filtering as baseline.

Like the Lorenz system test run, we observe a similar degradation in performance by any a priori filtering approach for a Gaussian noise level below 20%. Again, this comes down to the issue of over-smoothing. On the contrary, significantly enhanced performance is seen above this 20% threshold and much more so than in the ODE case. The ℓ_1 trend filter being the best performing singular filter, it still managed a successful identification rate of the dynamics at 31% whereas the no filtered models never succeed at the 75% noise mark. Average coefficient errors and wrong term numbers follow the general same trend, except for the average coefficient error actually being the lowest for the no filtered approach at 60%+ noise. We note once more, however, that at this point the models found without filtering never contain the correct terms so the metric becomes contextually of little importance.

Furthermore, while the Lorenz system showed hints of the advantages of a two-stage filtering design, their benefits become highly evident here for the viscous Burgers' PDE. Across all metrics, we see a constant bump in performance from passing the already filtered data through a secondary one. Resultingly, with the ℓ_1 method being the superior singular filter now, the $\ell_1$sav double filter shows the strongest performance overall. In a similar vein, the savℓ_1 approach enhances the standalone Savitzky-Golay filter's results, although not as majorly. These notable improvements from the two-stage filter designs can again be attributed to the smoothing of sharp corners and mitigation of jaggedness respectively as highlighted in Section 3.5.

For completion, we further investigate coefficient errors for this system as well by extracting metrics based only on correct identifications. Noting here that since the rates of successful identifications by the different designs visually vary in contrast to the Lorenz case, it is not possible to get a result where all methods are correct in the higher noise bands. Therefore, we instead only consider the first 10 runs and compute the error metrics based on these instead of searching for 10 instances where all are correct. Doing this for Gaussian noise levels of 25%, 50%, 75%, and 100%, where the variable u is referred to as $x0$, we obtain:

Target Dynamics: $(x0)' = 0.100x0_{11} - 1.000x0x0_1$

25%	**No Filter:** $(x0)' = 0.086x0_{11} - 0.973x0x0_1$	$[s = 4/10, E_c = 0.041]$
	Savitzky-Golay: $(x0)' = 0.114x0_{11} - 0.982x0x0_1$	$[s = 6/10, E_c = 0.032]$
	ℓ_1 Trend Filter: $(x0)' = 0.107x0_{11} - 0.984x0x0_1$	$[s = 9/10, E_c = 0.023]$
	savℓ_1 Two-Stage: $(x0)' = 0.114x0_{11} - 0.984x0x0_1$	$[s = 7/10, E_c = 0.030]$
	$\ell_1$sav Two-Stage: $(x0)' = 0.110x0_{11} - 0.988x0x0_1$	$[s = 9/10, E_c = 0.022]$
50%	**No Filter:** $(x0)' = -0.135x0_1 - 0.248x0x0_1 - 0.675x0x0x0_1$	
	Savitzky-Golay: $(x0)' = 0.107x0_{11} - 0.981x0x0_1$	$[s = 2/10, E_c = 0.026]$
	ℓ_1 Trend Filter: $(x0)' = 0.116x0_{11} - 0.992x0x0_1$	$[s = 5/10, E_c = 0.024]$
	savℓ_1 Two-Stage: $(x0)' = 0.105x0_{11} - 0.981x0x0_1$	$[s = 2/10, E_c = 0.024]$
	$\ell_1$sav Two-Stage: $(x0)' = 0.117x0_{11} - 1.001x0x0_1$	$[s = 5/10, E_c = 0.018]$
75%	**No Filter:** $(x0)' = 0.322x0 - 0.124x0x0 - 0.809x0x0_1 + 0.041x0x0x0_{11}$	
	Savitzky-Golay: $(x0)' = 0.115x0_{11} - 0.935x0x0_1$	$[s = 2/10, E_c = 0.080]$
	ℓ_1 Trend Filter: $(x0)' = 0.114x0_{11} - 0.953x0x0_1$	$[s = 3/10, E_c = 0.061]$
	savℓ_1 Two-Stage: $(x0)' = 0.117x0_{11} - 0.938x0x0_1$	$[s = 2/10, E_c = 0.079]$
	$\ell_1$sav Two-Stage: $(x0)' = 0.116x0_{11} - 0.962x0x0_1$	$[s = 4/10, E_c = 0.054]$
100%	**No Filter:** $(x0)' = -0.151x0x0 + 0.042x0_{11} - 0.896x0x0_1 - 0.003x0x0_{11}$	
	Savitzky-Golay: $(x0)' = 0.122x0 - 0.346x0x0 - 0.149x0_1 - 0.564x0x0_1$	
	ℓ_1 Trend Filter: $(x0)' = 0.098x0_{11} - 0.941x0x0_1$	$[s = 2/10, E_c = 0.061]$
	savℓ_1 Two-Stage: $(x0)' = 0.113x0_{11} - 1.065x0x0_1$	$[s = 1/10, E_c = 0.078]$
	$\ell_1$sav Two-Stage: $(x0)' = 0.104x0_{11} - 0.953x0x0_1$	$[s = 3/10, E_c = 0.051]$

Table 3: Average correctly extracted dynamics of viscous Burgers' PDE over 10 iterations for Gaussian noise strengths of 25%, 50%, 75%, and 100%. If a correct identification was not produced, an arbitrary choice is presented unless it would lead to a later correct identification with further processing, in which case the corresponding dynamics is given.

From this, we see additional evidence of what has previously been observed. First off, severely hindered performance from not using an a priori filter is seen already at the 50% mark for reconstructing the PDE, whereas the filters, particularly the ℓ_1 method, still does well. Most notably, the benefits of a two-stage design is further highlighted. Across all the coefficient error metrics, although minor, a constant reduction is seen from the data filtered twice in comparison to its respective first singular component. Furthermore, an instance at the 100% noise level is observed where the standalone Savitzky-Golay filter fails but its corresponding two-stage design savℓ_1 succeeds. Although this possibility was already evident from Figure 18, we note that the single filter was highly spurious and not even close to the true dynamics. This means the amount of jaggedness generated by the Savitzky-Golay algorithm at this high noise strength became particularly challenging for the weak SINDy formulation, but that the following ℓ_1 component has smoothed it out significantly. With this system not being chaotic [50] like that of the Lorenz system and the found coefficients of correct identifications being numerically closer to their true values, we are satisfied with these results without any further simulation.

6 Discussion and Conclusion

Having investigated the singular Savitzky-Golay and ℓ_1 filters with their corresponding savℓ_1 and $\ell_1$sav two-stage designs as a priori pre-processing steps for the weak SINDy formulation under increasing levels of Gaussian noise for both the Lorenz ODE system and viscous Burgers' PDE, several key trends have been identified. Referring back to our first aim of determining whether a single filter is even helpful as a start under this formulation, interesting findings emerged. Even though a Pareto criterion was utilized in selecting the filtering parameters to yield the most optimal trade-off between smoothing and retaining details for each method, reconstructing the Lorenz system showed no visually significant variation in successful identification rates when compared to the no filtering baseline except for harming the metric up to the 20% noise level. This result comes about due to the issue of over-smoothing, where in low noise bands the Pareto selection may mistake actual features of the data as noise when determining the best trade-off. In terms of coefficient errors, however, either filtering method managed to consistently start out performing the no filtered baseline at the 30% mark and up. Investigating this further considering only correct identifications, this gap became more pronounced. Consequently, with the Lorenz system being chaotic, either of the filtering approaches produced simulated dynamics in much closer vicinity to the true form for smaller simulation periods with the Saviztky-Golay filter being the strongest. In regards to the viscous Burgers' PDE, on the other hand, major improvements across the board were seen through a priori filtering. Like the Lorenz case, we also first note harm in the success rate below the 20% region here as well for the same reason. Above this band, however, great enhancement is seen particularly from the ℓ_1 trend filter where at the 50% noise level no success is observed from the no filtered baseline but the ℓ_1 and Savitzky-Golay approaches manages success rates of around 50% and 30% respectively. Coefficient errors, both for all identifications and only correct ones, saw strong improvements as well alongside the average number of incorrectly terms found.

When the qualitative outputs of the Pareto tuned single filters were investigated in Section 3.4, it was found that each of the filters started producing increasingly undesirable distortions

as the noise strength was heightened. For the Saviztky-Golay and ℓ_1 trend filter, these were either portions of jaggedness or sharp corners respectively. To tackle this, we proposed the savℓ_1 and $\ell_1$sav two-stage designs which efficiently eliminated the issues as documented in Section 3.5. This translated into visible improvements in the identification of the nonlinear dynamical systems studied as well. We found that for all metrics, although relatively minor, these designs provided a consistent bump in performance. We note that in the Lorenz ODE case where success rates remained stagnant, the coefficient errors improved particularly evident when examining correct identifications. In the Burgers' PDE domain, here both coefficient errors for all and correct extractions, together with success rates, saw a similar jump in enchantment. For instance, in the very high noise band of 70%+, the two-stage $\ell_1$sav design provided a stable 2.5%-4.0% increase in the correct identification rate to the already best performing singular ℓ_1 trend filter.

For practical real world application, our findings can be generalized into a few core concepts. To start, if an unknown system under study, either ODE or PDE-based, is known to be represented by data that does not exceed a Gaussian noise strength of 15-20% and the weak formulation of SINDy is to be used to reconstruct its dynamics, a priori filtering should not be used. If, however, it is expected to be above this, filtering is highly recommended particularly if a PDE structure is expected. Even if the system is believed to be represented by an ODE, however, it would still be beneficial with coefficient errors having still seen a constant reduction. Additionally, although we did not find conclusive evidence of either single filter being overall superior, we showed the augmentation that a two-stage design yielded in both cases. Therefore, although more computationally expensive, a two-stage a priori approach was ultimately stronger than its corresponding first singular component throughout our experiments, and is thus our final recommendation for adoption.

7 Limitations and Extensions

Throughout the course of this study, we confronted a spectrum of challenges including those tied to intricate computational demands. These challenges inevitably resulted in certain avenues being left unexplored to keep in line with the length requirement of this dissertation, offering good ground for subsequent research. Foremost, having only considered two systems within this investigation, the first point of interest would be in verifying whether the established trends hold, particularly for the more significantly varying PDE case. Additionally, although justification was given for employing Gaussian noise due to its uniqueness in nature, we have noted the existence of other noise types like impulsive and uniform. Thus, our conclusions are strictly valid only for noise following a Gaussian distribution, spotlighting another area of potential research. Moreover, even though studying a richer collection of standalone filters could provide insight into the most optimal performer, our study was centered around first verifying whether filtering is even appropriate and secondly the advantages of a two-stage design. Therefore, having verified the first aim, this can also be studied where we expect the findings of the two-stage approaches to hold.

Regarding technical decisions, we utilized a known set of constant regularization optimizer values for each problem domain in initializing the weak SINDy algorithm. Recognizing that this can also be viewed as a potential optimization problem, there may be certain parameter combinations that better align if it is known that a filter is to be applied. Having already established improvement with constant parameters, it is possible that this could strengthen a filter approach even further. Additionally, we address the point of having used a fixed polynomial degree n for the Savitzky-Golay filter for each system. This value was theoretically obtained based on the dynamics we needed to reconstruct, so although it is expected to yield the best performance, varying this value could give more information on the generally best singular filter and consequently the first component of preference in a two-stage design. Lastly, we reiterate once more that although a series of metrics were contextually used in deriving our conclusions, none of them are perfect due to the inherent intricacies involved with interpreting equation sets.

8 Bibliography

[1] Bongard, J. and Lipson, H. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*. 104(24), 9943-9948, 2007. doi: 10.1073/pnas.0609476104.

[2] Champion, K.P. et al. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*. 116(45), 22445-22451, 2019. doi: 10.1073/pnas.1906995116.

[3] Schmid, P.J. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*. 656, 5-28, 2010. doi: 10.1017/S0022112010001217.

[4] Mezić, I. Analysis of Fluid Flows via Spectral Properties of the Koopman Operator. *Annual Review of Fluid Mechanics*. 45, 357-378, 2013. doi: 10.1146/annurev-fluid-011212-140652.

[5] Brunton, S. et al. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*. 113(15), 3932-3937, 2016. doi: 10.1073/pnas.1517384113.

[6] Fasel, U. et al. Ensemble-SINDy: Robust sparse model discovery in the low-data, high-noise limit, with active learning and control. *Proceedings of the Royal Society A: Mathematical, Physical, and Engineering Sciences*. 478(2260), 2022. doi: 10.1098/rspa.2021.0904.

[7] Harrison, L.M. et al. Stochastic models of neuronal dynamics. *Philosophical Transactions of the Royal Society B: Biological Sciences*. 360(1457), 1075-1091, 2005. doi: 10.1098/rstb.2005.1648.

[8] Tassopoulos, P. and Protonotarios, Y. Brownian Motion & the Stochastic Behaviour of Stocks. *Journal of Mathematical Finance*. 12(1), 138-149, 2022. doi: 10.4236/jmf.2022.121009.

[9] França, T. et al. Feature engineering to cope with noisy data in sparse identification. *Expert Systems with Applications*. 188, 2022. doi: 10.1016/j.eswa.2021.115995.

[10] Cortiella, A. et al. A Priori Denoising Strategies for Sparse Identification of Nonlinear Dynamical Systems: A Comparative Study. *Journal of Computing and Information Science in Engineering*. 23(1), 2022. doi: 10.1115/1.4054573.

[11] Reinbold, P.A.K. et al. Using noisy or incomplete data to discover models of spatiotemporal dynamics. *Physical Review E*. 101(1), 2020. doi: 10.1103/PhysRevE.101.010203.

[12] Chasnov, J.R. (Hong Kong University of Science and Technology). 8.2: One-Dimensional Bifurcations. *LibreTexts Mathematics*. Accessed 29 September 2023. URL: https://math.libretexts.org/Bookshelves/Differential_Equations/Differential_Equations_(Chasnov)/08:Nonlinear_Differential_Equations/8.02:One-Dimensional_Bifurcations.

[13] Champion, K.P. et al. Discovery of Nonlinear Multiscale Systems: Sampling Strategies and Embeddings. *SIAM Journal on Applied Dynamical Systems*. 18(1), 312-333, 2019. doi: 10.1137/18M1188227.

[14] Vaddireddy, H. et al. Feature engineering and symbol regression methods for detecting hidden physics from sparse sensor observation data. *Physics of Fluids*. 32(1), 2019. doi: 10.1063/1.5136351.

[15] Champion, K.P. et al. A Unified Sparse Optimization Framework to Learn Parsimonious Physics-Informed Models From Data. *IEEE Access*. 8, 169259-169271, 2019. doi: 10.48550/arXiv.1906.10612.

[16] Rudy, S.H. et al. Data-driven discovery of partial differential equations. *Science Advances*. 3(4), 2017. doi: 10.48550/arXiv.1609.06401.

[17] Zheng, P. et al. A Unified Framework for Sparse Relaxed Regularized Regression: SR3. *IEEE Access*. 7, 1404-1423, 2018. doi: 10.48550/arXiv.1807.05411.

[18] Chartrand, R. Numerical Differentiation of Noisy, Nonsmooth Data. *International Scholarly Research Notices Applied Mathematics*. 2011, 2011. doi: 10.5402/2011/164564.

[19] Liu, C. A Brief Introduction to the Weak Form. *Comsol Blog*. 2014. Accessed 29 September 2023. URL: https://www.comsol.com/blogs/brief-introduction-weak-form/.

[20] Schaeffer, H. and McCalla, S.G. Sparse model selection via integral terms. *Physical Review E covering statistical, nonlinear, biological, and soft matter physics*. 96(2), 2017. doi: 10.1103/PhysRevE.96.023302.

[21] Messenger, D.A. and Bortz, D.M. Weak SINDy: Galerkin-Based Data-Driven Model Selection. *SIAM Journal on Multiscale Modeling and Simulation*. 19(3), 1474-1497, 2021. doi: 10.1137/20M1343166.

[22] Messenger, D.A. and Bortz, D.M. Weak SINDy for partial differential equations. *Journal of Computational Physics*. 443, 2021. doi: 10.1016/j.jcp.2021.110525.

[23] Savitzky, A. and Golay, M.J.E. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *ACS Journal on Analytical Chemistry*. 36(8), 1627-1639, 1964. doi: 10.1021/ac60214a047.

[24] Kim, S. et al. ℓ_1 Trend Filtering. *SIAM Review*. 51(2), 2019. doi: 10.1137/070690274.

[25] Hansen, P.C. The L-curve and its use in the numerical treatment of inverse problems. *Department of Mathematical Modelling, Technical University of Denmark*. 2005. Accessed 29 September 2023. URL: https://www.sintef.no/globalassets/project/evitameeting/2005/lcurve.pdf.

[26] Press, W.H. 14.8 Savitzky-Golay Smoothing Filters. *Numerical Recipes in Fortran 77 - The Art of Scientific Computing Second Edition*. 644-649, 1997. Accessed 29 September 2023. URL: http://phys.uri.edu/nigh/NumRec/bookfpdf/f14-8.pdf.

[27] Fan, J. et al. Data-driven bandwidth selection in local polynomial fitting: variable bandwidth and spatial adaptation. *Journal of the Royal Statistical Society - Series B*. 57(2), 371-394, 1995. Accessed 29 September 2023. URL: https://www.jstor.org/stable/2345968.

[28] Fan, J. and Gijbels, I. Adaptive order polynomial fitting: bandwidth robustification and bias reduction. *Journal of Computational and Graphical Statistics*. 4(3), 213-227, 1995. doi: 10.1080/10618600.1995.10474678.

[29] SciPy Software. scipy.signal.savgol_filter. Accessed 29 September 2023. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol_filter.html.

[30] Lohninger, H. Savitzky-Golay Filter - Mathematical Details. *Fundamentals of Statistics*. Accessed 29 September 2023. URL: http://www.statistics4u.info/fundstat_eng/cc_filter_savgol_math.html.

[31] KiyonoLab. The idea of the Savitzky-Golay filter. YouTube. Uploaded by KiyonoLab on 2 August 2022. Accessed 29 September 2023. URL: https://www.youtube.com/watch?v=1SvDZPvUo_I&.

[32] Diamond, S. and Boyd, S. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *Journal of Machine Learning Research*. 17(83), 2016. doi: 10.5555/2946645.3007036.

[33] Domahidi, A. et al. ECOS: An SOCP Solver for Embedded Systems. *2013 European Control Conference (ECC)*. 3071-3076, 2013. doi: 10.23919/ECC.2013.6669541.

[34] O'Donoghue, B. et al. Operator Splitting for Conic Optimization via Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications*. 169, 1042-1068, 2016. doi: 10.1007/s10957-016-0892-3.

[35] Rafaeilzadeh, P. et al. Cross-Validation. *Encyclopedia of Database Systems*. 532-538, 2019. doi: 10.1007/978-0-387-39940-9_565.

[36] LeCun, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. 86, 2278-2324, 1998. doi: 10.1109/5.726791.

[37] Boyat, A.K. and Joshi, B.K. A Review Paper: Noise Models in Digital Image Processing. *Signal & Image Processing: An international Journal (SIPIJ)*. 6, 2015. doi: 10.5121/sipij.2015.6206.

[38] Rosenhouse, G. The essence of noise in nature with reference to acoustics. *WIT Transactions on Ecology and the Environment*. 160, 3-13, 2012. doi: 10.2495/DN120011.

[39] Gundersen, G. Random Noise and The Central Limit Theorem. *Gregory Gundersen Blog*. 2019. Accessed 29 September 2023. URL: https://gregorygundersen.com/blog/2019/02/01/clt/.

[40] DynamicsLab. PySINDy: Differentiable SINDy modeling in Python. *GitHub*. 2023. Accessed 29 September 2023. URL: https://github.com/dynamicslab/pysindy.

[41] DynamicsLab. PySINDy Documentation: Examples. *pysindy*. 2023. Accessed 29 September 2023. URL: https://pysindy.readthedocs.io/en/latest/examples/index.html.

[42] Lorenz, E.N. Deterministic Nonperiod Flow. *Journal of Atmospheric Sciences*. 20(2), 130-148, 1963. doi: 10.1175/1520-0469(1963)020¡0130:DNF¿2.0.CO;2.

[43] Park, J. et al. Systematic comparison between the generalized Lorenz equations and DNS in the two-dimensional Rayleigh-Bénard convection. *AIP Publishing*. 31(7), 2021. doi: 10.1063/5.0051482.

[44] Thanoon, T.Y. and Al-Azzawi, S.F. Stability of Lorenz Differential System by Parameters. *IASJ Tikrit Journal of Pure Science*. 15(2), 118-122, 2010. Accessed 29 September 2023. URL: https://www.iasj.net/iasj/article/37704.

[45] SciPy Software. scipy.integrate.solve_ivp. Accessed 29 September 2023. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html.

[46] Bateman, H. Some recent researches on the motion of fluids. *American Meteorological Society Journals*. 43(4), 163-170, 1915. doi: 10.1175/1520-0493(1915)43¡163:SRROTM¿2.0.CO;2.

[47] Burgers, J.M. A Mathematical Model Illustrating the Theory of Turbulence. *Advanced in Applied Mathematics*. 1, 171-199, 1948. doi: 10.1016/S0065-2156(08)70100-5.

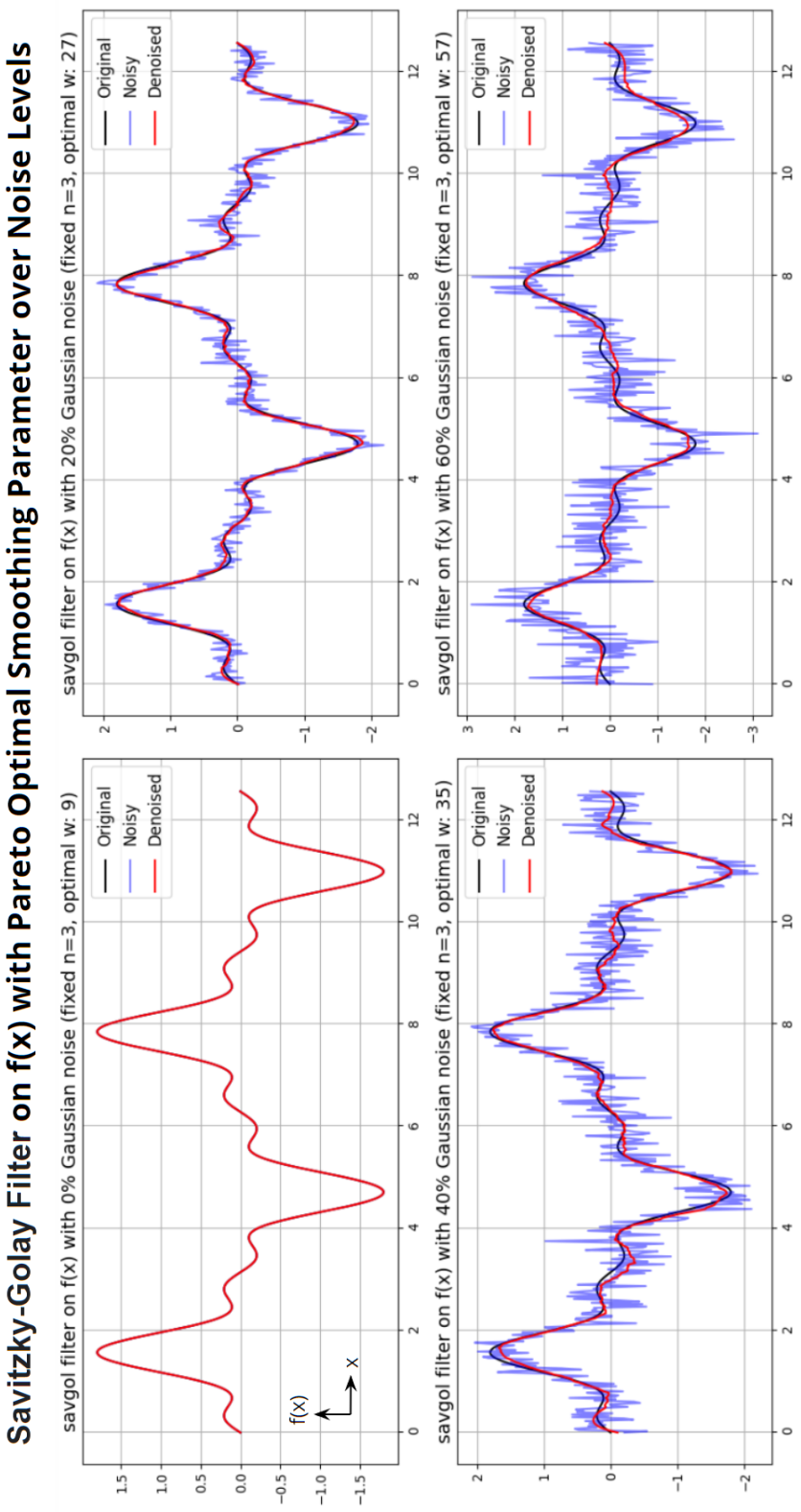
[48] Ramos, J.I. Shock waves of viscoelastic Burgers equations. *International Journal of Engineering Science*. 149, 2022. doi: 10.1016/j.ijengsci.2020.103226.

[49] Messenger, D.A. WSINDY_PDE. *GitHub*. 2020. Accessed 29 September 2023. URL: https://github.com/dm973/WSINDy_PDE/tree/master/datasets.

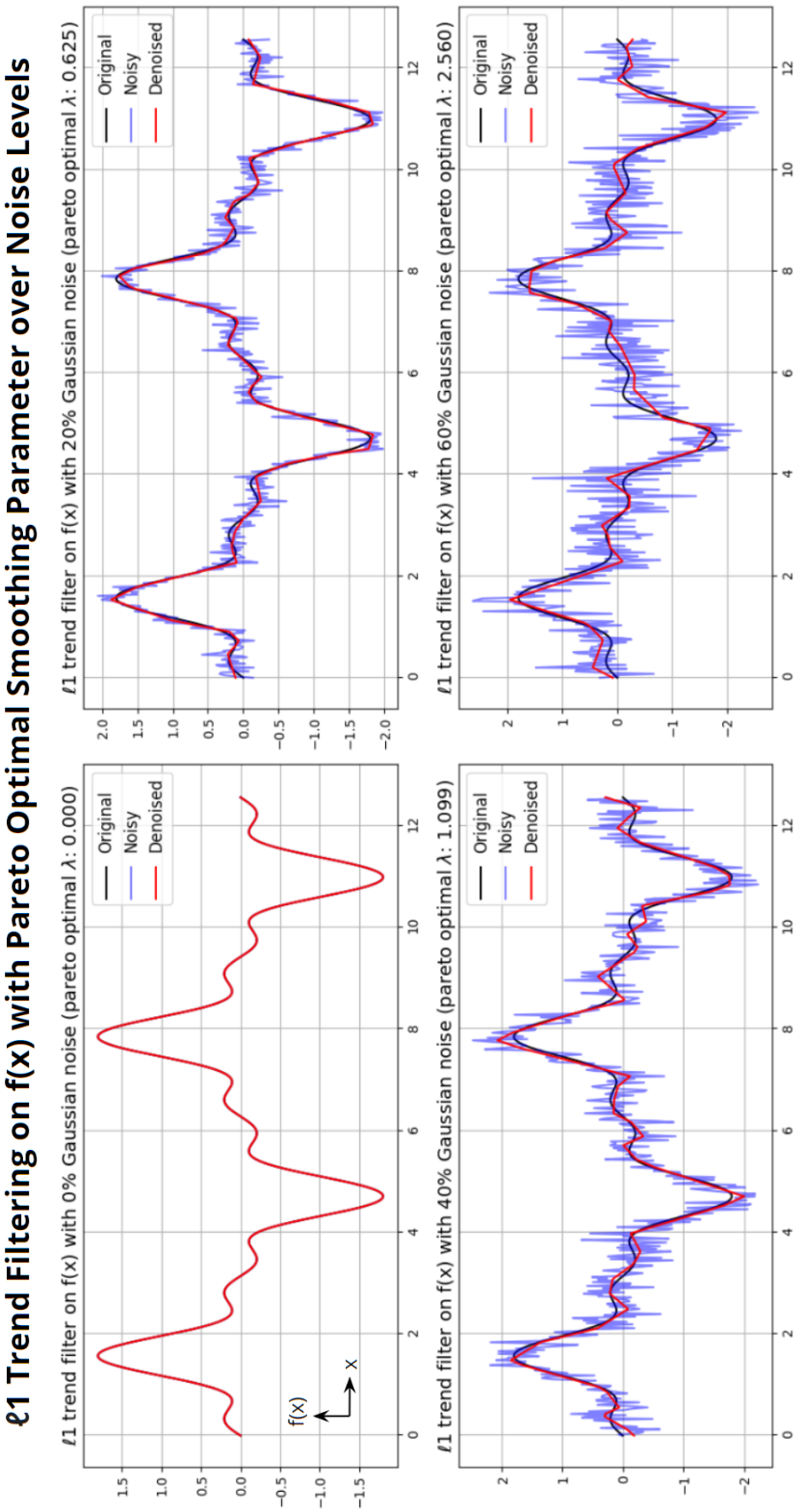
[50] Sabawi, M. Stability Study of Stationary Solutions of the Viscous Burgers equation. *Al-Rafidain Journal of Computer Science and Mathematics*. 4(1), 19-40, 2017. doi: 10.33899/csmj.2007.163993.

9 Appendix

9.1 Figure 5 Large-Scale



9.2 Figure 6 Large-Scale



9.3 Code and Data Repository

The complete code and experimental data used in this study can be accessed at the GitHub repository: https://github.com/oskar8991/msc_dissertation. Below we include key excerpts from the code central to the work. Versions of all major packages utilized were:

Package Name	Version Number
pysindy	1.7.5
scikit-image	0.19.3
scikit-learn	1.0.2
scipy	1.7.3
cvxpy	1.3.2
ecos	2.0.12
numpy	1.21.5
matplotlib	3.5.3

Table 4: List of Python packages and their versions used in the study.

Base Components:

```
# Gaussian noise addition
import numpy as np
from sklearn.metrics import mean_squared_error
def add_noise(data, percentage=0):
    rmse = mean_squared_error(u, np.zeros(data.shape), squared=False)
    data = data + np.random.normal(0, (percentage/100)*rmse, u.shape)
    return data

# Savitzky-Golay Filter
from scipy.signal import savgol_filter
def savgol_denoise_1D(data, window_length=5,poly_order=0):
    denoised_data = savgol_filter(data, window_length, poly_order)
    return denoised_data
```

```python
# l1 Trend Filter
import cvxpy as cp
def l1_trend_filter(y, lamb=0):
    n = len(y)
    x = cp.Variable(n)
    # Construct 2nd order difference matrix D2
    D = np.diff(np.eye(n), axis=0)
    D2 = np.diff(D, axis=0)
    # Define the objective
    objective = cp.Minimize(0.5 * cp.norm(x - y, 2)**2 +
                            lamb * cp.norm(D2 @ x, 1))
    # Construct the problem
    problem = cp.Problem(objective)
    # Apply ECOS solver
    try:
        problem.solve()
    except cp.SolverError:
        # Apply SCS solver if ECOS fails
        try:
            problem.solve(solver=cp.SCS)
        # No solution found
        except cp.SolverError:
            return None
    # Return the solution
    return x.value
```

Pareto selection of Savitzky-Golay W:

```
# Pareto selection of Savitzky-Golay window size
# Lorenz window range: 5 to 229, degree: 2
# Burgers window range: 5 to 99, degree: 3
def pareto_savgol_w(u_single):
    # Initialization
    windows = [i for i in range(5, 229, 2)]
    solution_norms = []
    residual_norms = []

    # Calculate norms for each window after filter application
    for w in windows:
        u_denoised = savgol_denoise_1D(u_single, w, 2)
        residual = u_single - u_denoised
        solution_norms.append(np.linalg.norm(u_denoised))
        residual_norms.append(np.linalg.norm(residual))

    solution_norms = np.array(solution_norms)
    residual_norms = np.array(residual_norms)
    # Find and extract point of most curvature
    curvature = np.gradient(np.gradient(solution_norms, residual_norms),
                            residual_norms)
    max_curvature_idx = np.argmax(np.abs(curvature))
    optimal_w = windows[max_curvature_idx]

    # Return optimal window size
    return optimal_w
```

Pareto selection of ℓ_1 Trend Filter λ:

```
# Pareto selection of l1 Trend Filter lambda
# Lorenz lambda range: 10^-5 to 10^2
# Burgers lambda range: 10^-5 to 10^1
def pareto_l1_lambda(u_single):
    # Initialization
    lambdas = np.logspace(-5, 2, 50)
    solution_norms = []
    residual_norms = []

    # Calculate norms for each window after filter application
    for lambda_reg in lambdas:
        u_denoised = l1_trend_filter(u_single, lambda_reg)
        residual = u_single - u_denoised
        solution_norms.append(np.linalg.norm(u_denoised))
        residual_norms.append(np.linalg.norm(residual))

    solution_norms = np.array(solution_norms)
    residual_norms = np.array(residual_norms)
    # Find and extract point of most curvature
    curvature = np.gradient(np.gradient(solution_norms, residual_norms),
                            residual_norms)
    max_curvature_idx = np.argmax(np.abs(curvature))
    optimal_lambda = lambdas[max_curvature_idx]

    # Return optimal lambda
    return optimal_lambda
```

Lorenz ODE System: Main Test Loop

```
# Lorenz Main Test Loop
# Add model printouts for correct equation extractions
# SINDy setup support from [https://pysindy.readthedocs.io/en/latest/
# examples/12_weakform_SINDy_examples/example.html#Test-weak-form-ODE-
# functionality-on-Lorenz-equation]

num_iterations = 500
noise_levels = [0,5,10,15,20,25,30,35,40,45,50,55,60,65,70]
true_coeffs_x = {'x': -10, 'y': 10}
true_coeffs_y = {'x': 28, 'y': -1, 'xz': -1}
true_coeffs_z = {'z': -2.666, 'xy': 1}
threshold = 0.0005
# Initialize training data
dt = 0.001
integrator_keywords = {}
feature_names = ['x', 'y', 'z']
t_train = np.arange(0, 8, dt)
x0_train = [-8, 8, 27]
t_train_span = (t_train[0], t_train[-1])
x_train_original = solve_ivp(lorenz, t_train_span, x0_train, t_eval=t_train,
                             **integrator_keywords).y.T

for p in noise_levels:
    # Initialize error metrics
    success_count = 0
    success_count_savgol = 0
    success_count_l1 = 0
    success_count_savl1 = 0
    success_count_l1sav = 0
    coeff_errors = []
    coeff_errors_savgol = []
    coeff_errors_l1 = []
    coeff_errors_savl1 = []
```

```
coeff_errors_l1sav = []
wrong_terms_list = []
wrong_terms_savgol_list = []
wrong_terms_l1_list = []
wrong_terms_savl1_list = []
wrong_terms_l1sav_list = []
print(f"Testing noise level: {p}")
for _ in range(num_iterations):
    # Add Gaussian noise corruption
    x_train = add_noise(x_train_original, p)
    # Initialize candidate library setup
    library_functions = [lambda x: x, lambda x, y: x * y, lambda x: x ** 2]
    library_function_names = [lambda x: x, lambda x, y: x + y,
                              lambda x: x + x]
    ode_lib = ps.WeakPDELibrary(library_functions=library_functions,
                            function_names=library_function_names,
                            spatiotemporal_grid=t_train,
                            is_uniform = True,
                            K=100,
                            include_bias=False) # no constant term
    # No filter model
    x_dot_integral = ode_lib.convert_u_dot_integral(x_train)
    model = ps.SINDy(feature_names=feature_names,
                optimizer=ps.STLSQ(),
                feature_library=ode_lib)
    model.fit(x_train, x_dot=x_dot_integral,t=dt)
    #print("no filter")
    #model.print()
    coeffs = model.coefficients()
    # Filtering
    num_vars = x_train.shape[1]
    x_train_savgol = np.zeros_like(x_train)
    x_train_l1 = np.zeros_like(x_train)
    x_train_savl1 = np.zeros_like(x_train)
```

```
x_train_l1sav = np.zeros_like(x_train)
for i in range(num_vars):
    x_pareto_w = lorenz_pareto_savgol_w(x_train[:, i])
    x_train_savgol[:, i] = savgol_denoise_1D(x_train[:, i],
                                             x_pareto_w, 2)
    x_pareto_lambda = lorenz_pareto_l1_lambda(x_train[:, i])
    x_train_l1[:, i] = l1_trend_filter(x_train[:, i], x_pareto_lambda)
for i in range(num_vars):
    savgol_pareto_lambda = lorenz_pareto_l1_lambda(x_train_savgol[:, i])
    x_train_savl1[:, i] = l1_trend_filter(x_train_savgol[:, i],
                                          savgol_pareto_lambda)
    l1_pareto_w = lorenz_pareto_savgol_w(x_train_l1[:, i])
    x_train_l1sav[:, i] = savgol_denoise_1D(x_train_l1[:, i],
                                            l1_pareto_w, 2)
# Fitting filtered models
x_dot_integral_savgol = ode_lib.convert_u_dot_integral(x_train_savgol)
x_dot_integral_l1 = ode_lib.convert_u_dot_integral(x_train_l1)
x_dot_integral_savl1 = ode_lib.convert_u_dot_integral(x_train_savl1)
x_dot_integral_l1sav = ode_lib.convert_u_dot_integral(x_train_l1sav)
model_savgol = ps.SINDy(feature_names=feature_names,optimizer=ps.STLSQ(),
                        feature_library=ode_lib)
model_savgol.fit(x_train_savgol, x_dot=x_dot_integral_savgol, t=dt)
coeffs_savgol = model_savgol.coefficients()
#print("savgol")
#model_savgol.print()
model_l1 = ps.SINDy(feature_names=feature_names,optimizer=ps.STLSQ(),
                    feature_library=ode_lib)
model_l1.fit(x_train_l1, x_dot=x_dot_integral_l1, t=dt)
coeffs_l1 = model_l1.coefficients()
#print("l1")
#model_l1.print()
model_savl1 = ps.SINDy(feature_names=feature_names,optimizer=ps.STLSQ(),
                       feature_library=ode_lib)
model_savl1.fit(x_train_savl1, x_dot=x_dot_integral_savl1, t=dt)
```

```
coeffs_savl1 = model_savl1.coefficients()
#print("savl1")
#model_savl1.print()
model_l1sav = ps.SINDy(feature_names=feature_names,optimizer=ps.STLSQ(),
                       feature_library=ode_lib)
model_l1sav.fit(x_train_l1sav, x_dot=x_dot_integral_l1sav, t=dt)
coeffs_l1sav = model_l1sav.coefficients()
#print("l1sav")
#model_l1sav.print()


timesteps = np.arange(x_train.shape[0])
labels = ['x', 'y', 'z']
all_terms = model.get_feature_names()
# Desired terms as per given equations
desired_terms_x = set(['x', 'y'])
desired_terms_y = set(['x', 'y', 'xz'])
desired_terms_z = set(['z', 'xy'])
# Helper function to get active terms for an equation
def get_active_terms(coeff_row, all_terms):
    return set([term for idx, term in enumerate(all_terms)
                if abs(coeff_row[idx]) > threshold])
# Extract active terms
active_terms_x = get_active_terms(coeffs[0], all_terms)
active_terms_y = get_active_terms(coeffs[1], all_terms)
active_terms_z = get_active_terms(coeffs[2], all_terms)
active_terms_x_savgol = get_active_terms(coeffs_savgol[0], all_terms)
active_terms_y_savgol = get_active_terms(coeffs_savgol[1], all_terms)
active_terms_z_savgol = get_active_terms(coeffs_savgol[2], all_terms)
active_terms_x_l1 = get_active_terms(coeffs_l1[0], all_terms)
active_terms_y_l1 = get_active_terms(coeffs_l1[1], all_terms)
active_terms_z_l1 = get_active_terms(coeffs_l1[2], all_terms)
active_terms_x_savl1 = get_active_terms(coeffs_savl1[0], all_terms)
active_terms_y_savl1 = get_active_terms(coeffs_savl1[1], all_terms)
active_terms_z_savl1 = get_active_terms(coeffs_savl1[2], all_terms)
```

```
active_terms_x_l1sav = get_active_terms(coeffs_l1sav[0], all_terms)
active_terms_y_l1sav = get_active_terms(coeffs_l1sav[1], all_terms)
active_terms_z_l1sav = get_active_terms(coeffs_l1sav[2], all_terms)
# Success Rates
if (active_terms_x == desired_terms_x and
    active_terms_y == desired_terms_y and
    active_terms_z == desired_terms_z):
    success_count += 1
if (active_terms_x_savgol == desired_terms_x and
    active_terms_y_savgol == desired_terms_y and
    active_terms_z_savgol == desired_terms_z):
    success_count_savgol += 1
if (active_terms_x_l1 == desired_terms_x and
    active_terms_y_l1 == desired_terms_y and
    active_terms_z_l1 == desired_terms_z):
    success_count_l1 += 1
if (active_terms_x_savl1 == desired_terms_x and
    active_terms_y_savl1 == desired_terms_y and
    active_terms_z_savl1 == desired_terms_z):
    success_count_savl1 += 1
if (active_terms_x_l1sav == desired_terms_x and
    active_terms_y_l1sav == desired_terms_y and
    active_terms_z_l1sav == desired_terms_z):
    success_count_l1sav += 1
# Coefficient Errors
coeff_error = 0
coeff_error_savgol = 0
coeff_error_l1 = 0
coeff_error_savl1 = 0
coeff_error_l1sav = 0
for term in desired_terms_x:
    idx = all_terms.index(term)
    coeff_error += abs(coeffs[0, idx] - true_coeffs_x[term])
    coeff_error_savgol += abs(coeffs_savgol[0, idx] - true_coeffs_x[term])
```

```
        coeff_error_l1 += abs(coeffs_l1[0, idx] - true_coeffs_x[term])
        coeff_error_savl1 += abs(coeffs_savl1[0, idx] - true_coeffs_x[term])
        coeff_error_l1sav += abs(coeffs_l1sav[0, idx] - true_coeffs_x[term])
    for term in desired_terms_y:
        idx = all_terms.index(term)
        coeff_error += abs(coeffs[1, idx] - true_coeffs_y[term])
        coeff_error_savgol += abs(coeffs_savgol[1, idx] - true_coeffs_y[term])
        coeff_error_l1 += abs(coeffs_l1[1, idx] - true_coeffs_y[term])
        coeff_error_savl1 += abs(coeffs_savl1[1, idx] - true_coeffs_y[term])
        coeff_error_l1sav += abs(coeffs_l1sav[1, idx] - true_coeffs_y[term])
    for term in desired_terms_z:
        idx = all_terms.index(term)
        coeff_error += abs(coeffs[2, idx] - true_coeffs_z[term])
        coeff_error_savgol += abs(coeffs_savgol[2, idx] - true_coeffs_z[term])
        coeff_error_l1 += abs(coeffs_l1[2, idx] - true_coeffs_z[term])
        coeff_error_savl1 += abs(coeffs_savl1[2, idx] - true_coeffs_z[term])
        coeff_error_l1sav += abs(coeffs_l1sav[2, idx] - true_coeffs_z[term])
    coeff_errors.append(coeff_error)
    coeff_errors_savgol.append(coeff_error_savgol)
    coeff_errors_l1.append(coeff_error_l1)
    coeff_errors_savl1.append(coeff_error_savl1)
    coeff_errors_l1sav.append(coeff_error_l1sav)
    # Wrong terms count
    wrong_terms = 0
    wrong_terms += len([term for term in active_terms_x if
                        term not in desired_terms_x])
    wrong_terms += len([term for term in active_terms_y if
                        term not in desired_terms_y])
    wrong_terms += len([term for term in active_terms_z if
                        term not in desired_terms_z])
    wrong_terms_savgol= 0
    wrong_terms_savgol += len([term for term in active_terms_x_savgol if
                               term not in desired_terms_x])
    wrong_terms_savgol += len([term for term in active_terms_y_savgol if
```

```
                                term not in desired_terms_y])
        wrong_terms_savgol += len([term for term in active_terms_z_savgol if
                                term not in desired_terms_z])
        wrong_terms_l1 = 0
        wrong_terms_l1 += len([term for term in active_terms_x_l1 if
                               term not in desired_terms_x])
        wrong_terms_l1 += len([term for term in active_terms_y_l1 if
                               term not in desired_terms_y])
        wrong_terms_l1 += len([term for term in active_terms_z_l1 if
                               term not in desired_terms_z])
        wrong_terms_savl1 = 0
        wrong_terms_savl1 += len([term for term in active_terms_x_savl1 if
                                  term not in desired_terms_x])
        wrong_terms_savl1 += len([term for term in active_terms_y_savl1 if
                                  term not in desired_terms_y])
        wrong_terms_savl1 += len([term for term in active_terms_z_savl1 if
                                  term not in desired_terms_z])
        wrong_terms_l1sav = 0
        wrong_terms_l1sav += len([term for term in active_terms_x_l1sav if
                                  term not in desired_terms_x])
        wrong_terms_l1sav += len([term for term in active_terms_y_l1sav if
                                  term not in desired_terms_y])
        wrong_terms_l1sav += len([term for term in active_terms_z_l1sav if
                                  term not in desired_terms_z])
        wrong_terms_list.append(wrong_terms)
        wrong_terms_savgol_list.append(wrong_terms_savgol)
        wrong_terms_l1_list.append(wrong_terms_l1)
        wrong_terms_savl1_list.append(wrong_terms_savl1)
        wrong_terms_l1sav_list.append(wrong_terms_l1sav)
    # Output metrics
    success_rate = (success_count / num_iterations) * 100
    success_rate_savgol = (success_count_savgol / num_iterations) * 100
    success_rate_l1 = (success_count_l1 / num_iterations) * 100
    success_rate_savl1 = (success_count_savl1 / num_iterations) * 100
```

```
success_rate_l1sav = (success_count_l1sav / num_iterations) * 100
print(f"Success rate using original data: {success_rate}%")
print(f"Success rate using savgol data: {success_rate_savgol}%")
print(f"Success rate using l1 data: {success_rate_l1}%")
print(f"Success rate using savl1 data: {success_rate_savl1}%")
print(f"Success rate using l1sav data: {success_rate_l1sav}%")
print(f"Avg coeff error using original data: {np.mean(coeff_errors)}")
print(f"Avg coeff error using savgol data: {np.mean(coeff_errors_savgol)}")
print(f"Avg coeff error using l1 data: {np.mean(coeff_errors_l1)}")
print(f"Avg coeff error using savl1 data: {np.mean(coeff_errors_savl1)}")
print(f"Avg coeff error using l1sav data: {np.mean(coeff_errors_l1sav)}")
print(f"Avg # wrong terms using original data: {np.mean(wrong_terms_list)}")
print(f"Avg # wrong terms using savgol data: {np.mean(wrong_terms_savgol_list)}")
print(f"Avg # wrong terms using l1 data: {np.mean(wrong_terms_l1_list)}")
print(f"Avg # wrong terms using savl1 data: {np.mean(wrong_terms_savl1_list)}")
print(f"Avg # wrong terms using l1sav data: {np.mean(wrong_terms_l1sav_list)}")
```

Viscous Burger's PDE: Main Test Loop

```
# Viscous Burgers Main Test Loop
# Add model printouts for correct equation extractions
# SINDy setup support from [https://pysindy.readthedocs.io/en/latest/
# examples/12_weakform_SINDy_examples/example.html#Test-weak-form-PDE-
# functionality-on-Burgers'-equation-with-20%-noise]

num_iterations = 500
noise_levels = [0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100]
threshold = 0.0005
true_coeffs = {'x0_11': 0.1, 'x0x0_1': -1.0}

for p in noise_levels:
    # Initialize error metrics
    success_count = 0
    success_count_savgol = 0
    success_count_l1 = 0
    success_count_savl1 = 0
    success_count_l1sav = 0
    coeff_errors = []
    coeff_errors_savgol = []
    coeff_errors_l1 = []
    coeff_errors_savl1 = []
    coeff_errors_l1sav = []
    wrong_terms_list = []
    wrong_terms_savgol_list = []
    wrong_terms_l1_list = []
    wrong_terms_savl1_list = []
    wrong_terms_l1sav_list = []
    print(f"Testing noise level: {p}")
    for _ in range(num_iterations):
        # Load data file
        data = loadmat("data/burgers.mat")
        time = np.ravel(data["t"])
```

```
x = np.ravel(data["x"])
u = np.real(data["usol"])
dt = time[1] - time[0]
dx = x[1] - x[0]
# Add Gaussian noise and reshape
u = add_noise(u, p)
u = np.reshape(u, (len(x), len(time), 1))
# Initialize candidate library setup
library_functions = [lambda x: x, lambda x: x * x]
library_function_names = [lambda x: x, lambda x: x + x]
# Define 2D spatiotemporal grid
X, T = np.meshgrid(x, time)
XT = np.asarray([X, T]).T
pde_lib = ps.WeakPDELibrary(
    library_functions=library_functions,
    function_names=library_function_names,
    derivative_order=2,
    spatiotemporal_grid=XT,
    is_uniform=True,
    K=1000,
)
# Filtering
u_savgol = np.empty_like(u)
u_l1 = np.empty_like(u)
u_savl1 = np.empty_like(u)
u_l1sav = np.empty_like(u)
u_pareto_lambda = burgers_pareto_l1_lambda(u)
u_pareto_w = burgers_pareto_savgol_w(u)
for j in range(u.shape[1]):
    u_savgol[:, j, 0] = savgol_denoise_1D(u[:, j, 0], u_pareto_w, 3)
for j in range(u.shape[1]):
    u_l1[:, j, 0] = l1_trend_filter(u[:, j, 0], u_pareto_lambda)
savgol_pareto_lambda = burgers_pareto_l1_lambda(u_savgol)
l1_pareto_w = burgers_pareto_savgol_w(u_l1)
```

```
for j in range(u.shape[1]):
    u_savl1[:, j, 0] = l1_trend_filter(u_savgol[:, j, 0], savgol_pareto_lambda)
for j in range(u.shape[1]):
    u_l1sav[:,j,0] = savgol_denoise_1D(u_l1[:, j, 0], l1_pareto_w, 3)
# Fit all models
optimizer = ps.SR3(
    threshold=0.1, thresholder="l0", tol=1e-8,
    normalize_columns=True, max_iter=1000
)
model = ps.SINDy(feature_library=pde_lib, optimizer=optimizer)
model_savgol = ps.SINDy(feature_library=pde_lib, optimizer=optimizer)
model_l1 = ps.SINDy(feature_library=pde_lib, optimizer=optimizer)
model_savl1 = ps.SINDy(feature_library=pde_lib, optimizer=optimizer)
model_l1sav = ps.SINDy(feature_library=pde_lib, optimizer=optimizer)
model.fit(u, quiet=True)
coeffs = model.coefficients()
#print("no filter")
#model.print()
model_savgol.fit(u_savgol, quiet=True)
coeffs_savgol = model_savgol.coefficients()
#print("savgol")
#model_savgol.print()
model_l1.fit(u_l1, quiet=True)
coeffs_l1 = model_l1.coefficients()
#print("l1")
#model_l1.print()
model_savl1.fit(u_savl1, quiet=True)
coeffs_savl1 = model_savl1.coefficients()
#print("savl1")
#model_savl1.print()
model_l1sav.fit(u_l1sav, quiet=True)
coeffs_l1sav = model_l1sav.coefficients()
#print("l1sav")
#model_l1sav.print()
```

```
# Extract active terms
all_terms = model.get_feature_names()
non_zero_indices = np.where(np.abs(coeffs) > threshold)[1]
active_terms = [all_terms[i] for i in non_zero_indices]
non_zero_indices_savgol = np.where(np.abs(coeffs_savgol) > threshold)[1]
active_terms_savgol = [all_terms[i] for i in non_zero_indices_savgol]
non_zero_indices_l1 = np.where(np.abs(coeffs_l1) > threshold)[1]
active_terms_l1 = [all_terms[i] for i in non_zero_indices_l1]
non_zero_indices_savl1 = np.where(np.abs(coeffs_savl1) > threshold)[1]
active_terms_savl1 = [all_terms[i] for i in non_zero_indices_savl1]
non_zero_indices_l1sav = np.where(np.abs(coeffs_l1sav) > threshold)[1]
active_terms_l1sav = [all_terms[i] for i in non_zero_indices_l1sav]
desired_terms = set(['x0_11', 'x0x0_1'])
# Success Rate
if set(active_terms) == desired_terms:
    success_count += 1
if set(active_terms_savgol) == desired_terms:
    success_count_savgol += 1
if set(active_terms_l1) == desired_terms:
    success_count_l1 += 1
if set(active_terms_savl1) == desired_terms:
    success_count_savl1 += 1
if set(active_terms_l1sav) == desired_terms:
    success_count_l1sav += 1
# Coefficient Errors
coeff_error = 0
coeff_error_savgol = 0
coeff_error_l1 = 0
coeff_error_savl1 = 0
coeff_error_l1sav = 0
for term in desired_terms:
    idx = all_terms.index(term)
    coeff_error += abs(coeffs[0, idx] - true_coeffs[term])
    coeff_error_savgol += abs(coeffs_savgol[0, idx] - true_coeffs[term])
```

```
            coeff_error_l1 += abs(coeffs_l1[0, idx] - true_coeffs[term])
            coeff_error_savl1 += abs(coeffs_savl1[0, idx] - true_coeffs[term])
            coeff_error_l1sav += abs(coeffs_l1sav[0, idx] - true_coeffs[term])
        coeff_errors.append(coeff_error)
        coeff_errors_savgol.append(coeff_error_savgol)
        coeff_errors_l1.append(coeff_error_l1)
        coeff_errors_savl1.append(coeff_error_savl1)
        coeff_errors_l1sav.append(coeff_error_l1sav)
        # Wrong terms count
        wrong_terms = [term for term in active_terms if
                       term not in desired_terms]
        wrong_terms_savgol = [term for term in active_terms_savgol if
                              term not in desired_terms]
        wrong_terms_l1 = [term for term in active_terms_l1 if
                          term not in desired_terms]
        wrong_terms_savl1 = [term for term in active_terms_savl1 if
                             term not in desired_terms]
        wrong_terms_l1sav = [term for term in active_terms_l1sav if
                             term not in desired_terms]
        wrong_terms_list.append(len(wrong_terms))
        wrong_terms_savgol_list.append(len(wrong_terms_savgol))
        wrong_terms_l1_list.append(len(wrong_terms_l1))
        wrong_terms_savl1_list.append(len(wrong_terms_savl1))
        wrong_terms_l1sav_list.append(len(wrong_terms_l1sav))
    # Output metrics
    success_rate = (success_count / num_iterations) * 100
    success_rate_savgol = (success_count_savgol / num_iterations) * 100
    success_rate_l1 = (success_count_l1 / num_iterations) * 100
    success_rate_savl1 = (success_count_savl1 / num_iterations) * 100
    success_rate_l1sav = (success_count_l1sav / num_iterations) * 100
    print(f"Success rate using original data: {success_rate}%")
    print(f"Success rate using savgol data: {success_rate_savgol}%")
    print(f"Success rate using l1 data: {success_rate_l1}%")
    print(f"Success rate using savl1 data: {success_rate_savl1}%")
```

```
print(f"Success rate using l1sav data: {success_rate_l1sav}%")
print(f"Avg coeff error using original data: {np.mean(coeff_errors)}")
print(f"Avg coeff error using savgol data: {np.mean(coeff_errors_savgol)}")
print(f"Avg coeff error using l1 data: {np.mean(coeff_errors_l1)}")
print(f"Avg coeff error using savl1 data: {np.mean(coeff_errors_savl1)}")
print(f"Avg coeff error using l1sav data: {np.mean(coeff_errors_l1sav)}")
print(f"Avg # wrong terms using original data: {np.mean(wrong_terms_list)}")
print(f"Avg # wrong terms using savgol data: {np.mean(wrong_terms_savgol_list)}")
print(f"Avg # wrong terms using l1 data: {np.mean(wrong_terms_l1_list)}")
print(f"Avg # wrong terms using savl1 data: {np.mean(wrong_terms_savl1_list)}")
print(f"Avg # wrong terms using l1sav data: {np.mean(wrong_terms_l1sav_list)}")
```