Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Sparse Identification of Symplectic Hamiltonian Dynamics for Predictive Modeling and Analysis

Nigel Bruce Khan

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Sparse Identification of Symplectic Hamiltonian Dynamics for Predictive Modeling and Analysis

Author:	Nigel Bruce Khan
Examiner:	Univ.-Prof. Dr. Eric Sonnendrücker
Scientific advisor:	Dr. Michael Kraus
Submission Date:	November 30th, 2023

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

November 30th, 2023 Nigel Bruce Khan

Acknowledgments

They say even small scientific progress is made by standing on the shoulders of giants. I had a lot of help climbing up to those shoulders. Firstly, I would like to thank my supervisor, Dr. Michael Kraus, for his kind-hearted guidance in academia and life, continuous support during this thesis, and willingness to work with a non-physicist. I want to thank my loving parents, who have always encouraged my academic pursuits and who unknowingly challenged me to better understand my thesis while trying to explain the topic to them. I thank my friends Carlos, Erfan, and Beril for their detailed feedback and care, even providing food and lodging and the occasional, much-needed push to keep me going. Finally, I would like to thank my closest friend and flower, Daria, who has stood by me through this whole endeavor and much more. I could not have managed this climb without all the help and love I received.

Abstract

This thesis addresses the challenge of identifying governing equations that preserve the symplectic structure for Hamiltonian systems in canonical conjugate coordinates. Employing the Sparse Identification of Nonlinear Dynamics (SINDy), a data-driven method for model discovery, our research introduces an extension tailored to Hamiltonian systems.

The methodology encompasses the joint discovery of parsimonious dynamical models and effective coordinates through the integration of sparse regression and an autoencoder. The principal outcome of this research is an enhanced iteration of SINDy adept at preserving symplectic structure while effectively capturing the dynamics inherent to Hamiltonian systems. Notably, the proposed method demonstrates the capability to concurrently unveil canonical coordinates, albeit within explicitly defined limitations. Models from classical physics, solid-state physics, and fluid mechanics substantiate the effectiveness of the approach in faithfully capturing Hamiltonian dynamics. However, the method's applicability encountered limitations in a celestial mechanics model. The methodology simultaneously successfully identified canonical conjugate coordinates, albeit within a constrained context.

This contribution aligns with the evolving landscape of data-driven discovery, offering a tool for predictive modeling and analysis within a formulation of mechanics. The novel SINDy extension offers a versatile tool for researchers engaged in data-driven modeling, scientific computing, and Hamiltonian physics.

Key words: SINDy Symplectic Structure Hamiltonian Dynamics Data-Driven Modeling

Contents

Acknowledgements vii

Abstract ix

1 Introduction 1

2 Background 3
2.1 Hamiltonian Systems . . . 3
2.1.1 Basics of Hamiltonian Mechanics . . . 4
2.1.2 Hamiltonian Formalism . . . 7
2.1.3 Symplectic Form . . . 8
2.2 Sparse identification of nonlinear dynamics (SINDy) . . . 9
2.3 Intrinsic Coordinate Identification . . . 12

3 Methodology 17
3.1 SINDy Algorithm . . . 17
3.1.1 SINDy with Unconstrained optimization . . . 20
3.2 Hamiltonian-SINDy Algorithm . . . 22
3.2.1 Hamiltonian-SINDy Without Gradient Data . . . 26
3.3 Intrinsic Coordinate Identification with SINDy . . . 28
3.4 Auto-Encoder-Hamiltonian-SINDy Algorithm . . . 31

4 Results 33
4.1 Classical SINDy . . . 33
4.1.1 Illustrative Example: Two-Dimensional Damped Linear Oscillator . . . 35
4.1.2 Chaotic System: Lorenz Attractor . . . 36
4.1.3 Nonlinear PDE: Fluid wake behind a cylinder . . . 37
4.2 Hamiltonian SINDy . . . 40
4.2.1 Nonlinear Oscillator Model . . . 41
4.2.2 Toda Lattice . . . 42
4.2.3 Point Vortex . . . 44
4.2.4 Solar System . . . 46
4.3 Intrinsic Coordinate Identification . . . 47
4.3.1 Damped Linear Oscillator with an Autoencoder . . . 48
4.3.2 Lorenz system with an Autoencoder . . . 49
4.4 Autoencoder Hamiltonian SINDy . . . 53

5 Conclusion **57**
5.0.1 Outlook 57

Appendix **61**
.1 Coefficients Tables 61

Bibliography **61**

1 Introduction

In the landscape of scientific computing and physics, where the extraction of meaningful insights from dynamic processes relies on the interplay of mathematical models and data, the quest for structure preservation is an ongoing endeavor. At the forefront of this pursuit are methods that bridge the realms of model reduction and data-driven discovery, focusing on preserving the inherent structures and symmetries governing complex systems. This thesis navigates this interdisciplinary terrain, developing a structure-preserving method to transform data into intrinsic coordinates while extracting sparse governing equations.

Model reduction, a cornerstone in computational physics, endeavors to distill essential dynamics from intricate mathematical models. Sparse Identification of Nonlinear Dynamics (SINDy), an increasingly influential methodology, has proven adept at identifying sparse representations of governing equations from time-series data. However, the challenge lies in extending its capabilities beyond mere identification to structure preservation—an essential facet in ensuring the fidelity of reduced models to the underlying physics.

Central to this work is incorporating Hamiltonian system dynamics into the SINDy framework while concurrently preserving the symplectic structure—an essential attribute of conservative systems. The challenge is multifaceted, requiring the identification of sparse dynamical models and preserving symplectic geometry, a task traditionally elusive in data-driven reduction methods.

The concept of structure preservation is particularly crucial in hyper-reduction, where the objective is to create compact yet faithful representations of complex systems. Hyper-reduction techniques aim to distill a system's essential features and maintaining fundamental structures, such as symplectic form there would be crucial. This thesis takes a step in this direction by addressing the intricate problem of conjugate coordinate transformation solely through the universal approximation power of neural networks and a structure preservation SINDy-based model.

Integrating an autoencoder into the SINDy paradigm represents a novel approach to addressing this challenge. By leveraging an autoencoder, we aim to discover canonical conjugate coordinates that not only facilitate the identification of parsimonious dynamical models but also preserve the symplectic form inherent in Hamiltonian systems. This unique combination allows data conversion to canonical conjugate coordinates, enabling seamless integration of data-driven model reduction while preserving Hamiltonian structures. As we delve deeper into the subsequent chapters, the thesis comprehensively explores this methodology, elucidating its theoretical underpinnings, algorithmic intricacies, and empirical validations. The ultimate goal is to contribute to advancing structure-preserving

hyper-reduction methodologies, enhancing our capacity to distill essential dynamics from vast datasets while faithfully preserving the symmetries and structures that define the underlying physical systems.

2 Background

2.1 Hamiltonian Systems

Hamiltonian mechanics provides a theoretical framework for describing dynamic systems, similar to Newtonian or Lagrangian mechanics. Although Newton's second law was a revolutionary equation for building mathematical models, it is often hard to solve when the system is complicated or has many interacting objects. Hamiltonian mechanics strength is shown when we tackle problems with many degrees of freedom, like in celestial mechanics, where having more than two bodies potentially gives rise to chaotic behavior. Other areas where this framework shines are condensed matter physics, fluid simulations, and many-body quantum mechanics [17]. This section will outline some of the advantages of the Lagrangian and Hamiltonian approaches over the Newtonian formulation and the overall advantages of the Hamiltonian framework.

One of the main features of Hamiltonian and Lagrangian mechanics is their conformity to Hamilton's Principle of least action, which states:

The path followed by a system that takes it from time t_1 to t_2 is the one that makes the action $S = \int_{t_1}^{t_2} L(q_i, \dot{q}_i, t)\, dt$, stationary [41].

Here, S is the action, L is the Lagrangian, and q_i and $\dot{q}_i$ are the generalized coordinates and their generalized velocities. This elegant principle reveals that although there could be infinitely many ways a system can move from one point to another, it always chooses the path that gives this integral's minimum, maximum, or saddle point. The solution to this integral is found through the Euler-Lagrange Equation 2.8 below, for which one can choose any set of coordinates q_i. For example, in a pendulum system, one can choose the angle with the vertical as the coordinate, or choose polar coordinates for an object moving on a sphere, or spherical coordinates for systems with central forces [41]. This freedom to choose the coordinates while solving equivalent Euler-Lagrange formulations is one of the advantages of the Lagrangian and Hamiltonian frameworks because, for certain systems, it is possible to eliminate the explicit appearance of constraint forces by choosing a specific coordinate system.

Lagrangian and Hamiltonian formulations simplify identifying conserved quantities in dynamical systems while being equivalent to the Newtonian formulation. Lagrangian mechanics is contained in Hamiltonian mechanics as a special case [2], and we can reach the Hamiltonian formulation by using the Legendre transform on the Lagrangian formulation.

When the Lagrangian does not explicitly depend on time, i.e., $\frac{\partial L}{\partial t} = 0$, as in conservative systems, we can apply the Legendre transform to it, as below:

$$H(q_i, p_i, t) = \sum_i \dot{q}_i p_i - L(q_i, \dot{q}_i, t), \tag{2.1}$$

The independent variable changes from generalized velocity in the Lagrangian to generalized momentum in the Hamiltonian, often a conserved quantity. The conjugate momentum p_i corresponding to the generalized coordinate q_i is given by $p_i = \frac{\partial L}{\partial q_i}$, [41]. This formula is used while deriving the Hamiltonian from the Lagrangian in section 2.1.1 below. The Hamiltonian is known to be more physical than the Lagrangian as it constitutes the system's total energy, whereas the Lagrangian does not have a clear physical meaning.

Changing to the Hamiltonian formulation allows dynamics to be analyzed in phase space. The phase space gives a complete description of the system's state in terms of coordinates and momentum [41] because the system will follow a determined path in phase space, where governing equations can yield its phase space location at any given time. According to Liouville's theorem, a system's phase space volume is constant along its trajectory [41]. Consequently, if a many-particle system is now represented in phase space, we can see precisely how a change in one particle's momentum or position will affect the entire system. The Lagrangian state space (q, $\dot{q}$) has no theorem corresponding to Liouville's theorem for Hamiltonian systems [41]. This is one of the essential advantages the Hamiltonian approach has over the Lagrangian approach.

The evolutionary behavior of dynamical systems after a long time is a complex problem in mathematics [24]. Fortunately, one way to predict long-time dynamics is through symplectic integrators applied to Hamiltonian systems. By imposing the symplectic structure on numerical methods, phase space volume can be preserved, allowing for more accurate long-time predictions [20]. Moving in the direction of the symplectic gradient of H keeps the output exactly constant, compared to the normal gradient, which gives the direction of the fastest change [11]. Symplectic structure preservation has also been used to develop data-driven approaches for model prediction as seen in [7, 11, 24] and other works.

Although all three formulations, Newtonian, Lagrangian, and Hamiltonian, have certain advantages and disadvantages. In the scope of this paper, we will work on predicting Hamiltonian system dynamics while having symplectic form preservation built into the method.

2.1.1 Basics of Hamiltonian Mechanics

This section will discuss Hamiltonian mechanics in more detail by deriving the Hamiltonian. When introducing the Hamiltonian SINDy method later, it will be essential to understand the structure and constraints of Hamiltonian systems. It also helps in analyzing results and proving the accuracy of the algorithm.

As we saw before, Lagrangian mechanics is more closely related to Hamiltonian mechanics, leading to it more naturally than Newtonian mechanics through the Euler-Lagrange

equation and the Legendre transformation. The Hamiltonian will be derived from the Lagrangian viewpoint here. This derivation will explain how the Hamiltonian comes about and provide insight into how the two frameworks differ.

The Lagrangian is centered on the operator function L, which, for simple cases, is a function of the difference between the kinetic T and potential energy U. It is a function of the n generalized coordinates $q_1 ... q_n$ and the generalized velocities of these coordinates, $\dot{q}_1 ..., \dot{q}_n$:

$$L(q_1 ... q_n, \dot{q}_1 ... \dot{q}_n) = T - U \tag{2.2}$$

The q_1,...,q_n, and $\dot{q}_1$,...,$\dot{q}_n$ variables define a coordinate in $2n$-dimensional configuration space. When these n variables are taken together with a set of initial conditions, a unique orbit through state space can be determined for the system using the second-order Euler-Lagrange equations. Hence, we want to find a function that tells us how our system will progress if we start a time t_0 from a certain configuration. Taking the extremal principle from the calculus of variations and applying it to each degree of freedom in the system, we get the Euler-Lagrange equation [2], telling us how the system moves in time. We will derive the Euler-Lagrange equation below because it helps us understand where the Hamiltonian equations come from.

We start with the general form of the variational problem that represents the action integral [2]:

$$S = \int_{t_1}^{t_2} f(q(t), \dot{q}(t), t)\, dt, \tag{2.3}$$

We want to find a function f that keeps the action integral stationary to conform to the least action principle, i.e., $dS/df = 0$, as seen in the definition of the principle in section 2.1. We need the function q for this. Hence, we first assume that we have an approximation of it, expressed as $q(t) = q'(t) + \alpha h(t)$, where $q'(t)$ is the correct function, $h(t)$ is a function that is the difference between the correct solution $q(t)'$ and the approximate solution $q(t)$, and α is its amplitude. All three functions, q, q', and h, are assumed to pass through the time points t_1 and t_2, the limits of the integration. Since only the correct function q' will minimize S, we change the criteria for minimizing S. We need $\alpha = 0$ because S will be zero when $q = q'$. Hence, we want $dS/d\alpha = 0$. We can then rewrite S:

$$S(\alpha) = \int_{t_1}^{t_2} f(q(t), \dot{q}(t), t)\, dt = \int_{t_1}^{t_2} f(q' + \alpha h, \dot{q}' + \alpha \dot{h}, t)\, dt \tag{2.4}$$

From fundamental calculus, we know that the function $S(\alpha)$ needs to be minimized at $\alpha = 0$. The function's gradient will be zero at that point, meaning that S will become stationary, as required. Following this logic, we get the equation here:

$$\frac{dS}{d\alpha} = \frac{d}{d\alpha} \int_{t_1}^{t_2} f(q(t), \dot{q}(t), t)\, dt = \frac{d}{d\alpha} \int_{t_1}^{t_2} f(q' + \alpha h, \dot{q}' + \alpha \dot{h}, t)\, dt = 0 \tag{2.5}$$

The equation above is true for any path q'(t). Simplifying the derivative, then, we get:

$$\frac{dS}{d\alpha} = \int_{t_1}^{t_2} \frac{\partial f}{\partial \alpha}\, dt = \int_{t_1}^{t_2} \left(h\frac{\partial f}{\partial q} + \dot{h}\frac{\partial f}{\partial \dot{q}} \right) dt = 0 \tag{2.6}$$

We use integration by parts on the term $\int_{t_1}^{t_2} \dot{h}(t)\frac{\partial f}{\partial \dot{q}} dt$. At the boundaries of the integration i.e. t_1 and t_2, $h(t) = 0$ because all three functions $q(t)$, $q'(t)$ and $h(t)$ pass through those points, which means at t_1 and t_2, $q(t) = q'(t)$. Hence, $h(t) = 0$ at those points. Substituting the integration by parts results back into the equation above we get:

$$\frac{dS}{d\alpha} = \int_{t_1}^{t_2} h(t) \left(\frac{\partial f}{\partial q} - \frac{d}{dt}\frac{\partial f}{\partial \dot{q}} \right) dt = 0 \tag{2.7}$$

As shown before, the condition above is true for any choice of $h(x)$. That means $h(x)$ is not necessarily equal to zero in the above equation. This statement implies that the integrand must be zero. If we rewrite f as L in the integrand, we get what is called the Euler-Lagrange Equation:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = 0 \qquad \text{Euler-Lagrange Equation.} \tag{2.8}$$

We can see that the action re-written as $S = \int_{t_1}^{t_2} L(q, \dot{q}, t)dt$ is extremised if and only if the Euler-Lagrange equation is satisfied. In other words, the system's trajectory q is subject to the Euler-Lagrange equation. This equation is a system of n second-order equations, depending on $2n$ constants. The $2n$ conditions $q(t_0) = q_0$ and $q(t_1) = q_1$ are used to find these constants. The Euler-Lagrange equation is also independent of the coordinate system.

By substituting the Lagrangian given by $L = T - U$ into Equation 2.8 above, we obtain the concrete form of the differential equation that can be solved to get the equations of motion. L is the difference between the kinetic and potential energies, and the proof is as follows. Since $U = U(q)$, only depends on position, and $T = \sum m_i \dot{q}_i^2/2$, only depends on velocity, then $d/dt(\partial L/\partial \dot{q}_i) = d/dt(\partial T/\partial \dot{q}_i) = m_i \ddot{q}_i$ and $\partial L/\partial q_i = -\partial U/\partial q_i$. The first term is mass times acceleration, and the second is the equivalent force. Thus, the Euler-Lagrange formula in Equation 2.8 is true [2]. Next, we discuss the Legendre Transform.

An excellent description of the Legendre Transform can be found here: C. E. Mungan, "Legendre Transforms for Dummies," U.S. Naval Academy, Annapolis, MD [33]. This transformation converts one set of variables to another conjugate set of variables while preserving the underlying information. Changing the independent variables from position and velocity to position and conjugate momenta reveals more profound insights into symmetries in the equations of motion. It brings out the conservation of the canonical momenta [33].

If we apply the Legendre transform to the Lagrangian $L(q, \dot{q})$, we obtain the Hamiltonian:

$$H(q,p) = \sum_{i=1}^{n} p_i \dot{q}_i - L(q,\dot{q}) \tag{2.9}$$

To show that H is the total energy, we expand Equation 2.9, for the special case below:

$$H(q,p) = (mv)(v) - (\frac{1}{2}mv^2 - U) = \frac{1}{2}mv^2 + U = T + U \tag{2.10}$$

We see from this example that the Hamiltonian is the total energy of a system.

The Legendre Transform converted the variables from q and $\dot{q}$ to q and p, where p is the conjugate momentum of the generalized velocity $\dot{q}$. In the Hamiltonian formalism we obtain 1^{st} order equations of the variables $q_1, ..., q_n, p_1, ..., p_n$, consisting of n generalized positions and n conjugate momenta. These coordinates define a unique point in a 2n-dimensional phase space, completely defining a system's state. Starting the system from an initial condition in phase space, Hamilton's equations can determine a unique trajectory for the system's evolution in this space. The dynamics preserve phase space volume and other geometric properties [2]. In numerical applications, it is advantageous to preserve these properties as they will be important when we try to analyze the accuracy of our Hamiltonian SINDy method. Below, we formulate the dynamic equations of the Hamiltonian, which we will use to identify the governing equations of a system through data and a library of functions in Hamiltonian-SINDy.

2.1.2 Hamiltonian Formalism

We will derive the equations of motion of the Hamiltonian here, i.e., the derivatives of the Hamiltonian with respect to its arguments q and p. These are equal to $\frac{dq}{dt}$ and $\frac{dp}{dt}$, which tell us how the system moves in time, in phase-space. In this formulation, we will assume that neither the Lagrangian nor the Hamiltonian have an explicit time dependence. Let us take the derivative of the Hamiltonian in two ways. First, we express the total differential of the Hamiltonian, the derivative of H with respect to q and p as:

$$dH = \frac{\partial H}{\partial p}dp + \frac{\partial H}{\partial q}dq \tag{2.11}$$

From the Legendre transform of the Lagrangian function $H = p\dot{q} - L$, we know that the derivative above equals the total differential of $p\dot{q} - L$. We set the Lagrangian to $L = T - U$, i.e., the difference between the kinetic and potential energies. Let us define $U = U(q)$, and the kinetic energy as $T = \sum m_i \dot{q}_i^2/2$, then $\partial L/\partial \dot{q}_i = \partial T/\partial \dot{q}_i = m_i \dot{q}_i = p$. From these equations, we see that $p = \partial L/\partial \dot{q}$, and if we take the derivative of $H(q,p)$ with respect to its variables q, p, we can conclude that:

$$dH = \dot{q}dp - \frac{\partial L}{\partial q}dq \tag{2.12}$$

Since both equations for dH above must be the same, it can be concluded that:

$$\dot{q} = \frac{\partial H}{\partial p}, \qquad \frac{\partial H}{\partial q} = -\frac{\partial L}{\partial q} = -\dot{p} \tag{2.13}$$

We get the second equation from applying Euler-Lagrange's equations to $\partial L/\partial \dot{q}$ [2], which is why it was essential to derive it earlier. Neither the Lagrangian nor the Hamiltonian depends explicitly on time in conservative systems, so $\frac{\partial H}{\partial t} = -\frac{\partial L}{\partial t} = 0$

Putting everything together, we get the Hamiltonian equations of motion for a one-dimensional system.

$$\dot{q} = \frac{\partial H}{\partial p}, \qquad \dot{p} = -\frac{\partial H}{\partial q} \tag{2.14}$$

Here, one can see that unlike in the Lagrangian formalism, where we had a single second-order differential equation with the variable q, in the Hamiltonian method, we have *two first-order* differential equations with variables q and p. This separation into two sets of equations simplifies the analysis of complex systems and makes the representation of the system's phase space state easier. Our discussion has so far (implicitly) assumed that we are considering a one-dimensional system with only one q-component and one p-component. However, all derivations generalize straightforwardly to the case of arbitrary dimensions. With all these conditions in place, we can derive the Hamiltonian vector field equations for n dimensions as

$$\dot{q_i} = \frac{\partial H}{\partial p_i}, \qquad \dot{p_i} = -\frac{\partial H}{\partial q_i} \qquad \text{for all } i = 1, ..., n. \tag{2.15}$$

In the above derivation, we have also assumed that H is conserved, i.e., does not change explicitly with time (t). The total time derivative of H is the rate of change of H depending on the change in the coordinates $q_1 ..., q_n, p_1, ..., p_n$ with time. This total derivative is equal to zero without H explicitly depending on time because each of the terms $\frac{\partial H}{\partial p_i}$ and $\frac{\partial H}{\partial q_i}$ would cancel out in the sum [2], as the total energy H should be constant.

Now that the description and derivation of Hamiltonian Mechanics is complete, we can describe an important feature that will be used to ensure the SINDy method works for Hamiltonian systems while retaining its structure. This property is that a Hamiltonian matrix is symplectic, which will be briefly described below.

2.1.3 Symplectic Form

The Hamiltonian SINDy method uses information about the symplectic form of the Hamiltonian to identify the underlying dynamics of the system. A characteristic feature of Hamiltonian dynamics is that the Hamiltonian flow φ_t, i.e., the trajectory in phase space, is symplectic. This means that the derivative $\varphi_t' = \partial \varphi_t / \partial (q, p)$ must satisfy the following property:

$$\varphi_t'(q,p)^T J \varphi_t'(q,p) = J \quad \text{with } J = \begin{pmatrix} 0 & I_d \\ -I_d & 0 \end{pmatrix} \tag{2.16}$$

I_d is the identity matrix with dimensions equal to d, with the matrix $\varphi_t'(q,p) \in \mathbb{R}^{2d\times 2d}$. J is called the canonical skew-symmetric matrix. Preservation of phase space volume is a consequence of symplecticity, which in turn ensures the Hamiltonian and total energy in a conserved system are constant, as was pointed out in 1899 by Poincare[18]. Many Hamiltonian systems are not integrable, and numerical methods such as symplectic integrators are necessary to approximate their trajectories over time. These integrators enforce the symplectic structure and energy conservation by maintaining the phase space volume. If we write the Hamiltonian system in a compact form as $z = (q,p)^T$ and

$$J = \begin{pmatrix} 0 & I_d \\ -I_d & 0 \end{pmatrix}$$

Then, the system can be expressed as:

$$\frac{d}{dt}z = J\nabla_z H(z) \tag{2.17}$$

Evaluating the long-time behavior of dynamical systems is a well-known difficulty in mathematics. Situations where dynamics exhibit chaotic behavior or explode are often encountered. Fortunately for Hamiltonian systems, imposing the symplectic structure on numerical methods through Equation 2.16 can alleviate the problem [29].

In our *Hamiltonian-SINDy* method, we construct a parametrization of a Hamiltonian vector field that has the symplectic structure built into it. The structure ensures that the Hamiltonian properties are conserved no matter which dynamic equations are discovered. Now, we will discuss the method chosen to solve our problem of dynamics discovery.

2.2 Sparse identification of nonlinear dynamics (SINDy)

Traditionally, deriving governing equations is based on underlying first principles, such as conservation laws, symmetries, or universal laws. However, in many complex systems, governing equations are only partially known, and it is impossible or challenging to derive them from universal laws and first principles. Many systems, such as climate science or plasma physics, have rich time-series sensor and measurement data. Arising from this is the trend of data-driven model discovery, to which intense research effort is devoted. A major conflict in this discovery is the balance between a model's descriptive capabilities, complexity, and accuracy [6].

A parsimonious model is one that achieves the desired prediction accuracy with as few variables as possible. Parsimonious models are well balanced for accuracy, complexity, and descriptive power, capturing essential interaction with the fewest terms. Leveraging convex optimization ensures that these methods scale favorably to various large-scale problems. The SINDy method relies on the assumption that most physical systems have only a few relevant basis terms that define their dynamics, making the set of equations sparse in a high-dimensional nonlinear function space [5].

Similarly, other methods try to reproduce the dynamics from data. We will discuss some of them here to show how SINDy stands out. A popular data-driven approach utilizes Koopman Operator Theory to identify transformations from nonlinear to linear coordinates. If linear dynamics are found, closed-form solutions are acceptable, which allow for the analysis, prediction, and control of such systems [4]. However, linear models cannot capture the complete behavior of many nonlinear systems, limiting their accuracy and long-term predictive capability. Another method known as Dynamic Mode Decomposition (DMD) relies only on measurement data, not information about the governing equations. Even though advances have been made by combining it with Koopman operator theory, it can still not pinpoint which nonlinear functions are correct, so the form of the dynamics must be assumed [4]. Neural network methods can capture complex nonlinear dynamics and retain specific desirable properties from systems like total energy conservation and stability. However, these methods cannot guarantee the output of true dynamic equations. They usually need help retaining all the properties expected from the system, such as symplectic structure, and typically lead to models that are hard to interpret. They also require a more significant computational effort to train them than other methods because of the large data requirements for solutions with small errors [5, 23]. While maintaining their underlying properties, the SINDy method balances accuracy, effort, and interpretability well to find the correct dynamic equations in Newtonian, Lagrangian, and Hamiltonian frameworks.

SINDy is a sparse nonlinear regression method that aims for interpretability while finding parsimonious natural laws. It builds on symbolic regression but avoids over-fitting by adding a Pareto front as a parsimony constraint. A Pareto front is a dramatic change in the accuracy versus complexity curve parameterized by a variable λ [5]. SINDy makes it possible to find a system's relevant physics by identifying the governing equations using gradient and state data and some system knowledge in the form of a dictionary of basis functions. It uses data collected using simulations or experiments, assuming the data is noisy [5]. A nonlinear function that can represent the vector field of the system is what the method is trying to find:

$$\dot{x}(t) = f_\theta(x(t)). \tag{2.18}$$

The state of the system at a time t is represented by a vector $x(t) = [x_1(t) \quad x_2(t) \quad ... \quad x_n(t)]^T \epsilon \mathbb{R}^n$, with n states. The nonlinear function $f_\theta(x(t))$ above represents the vector field of the dynamic equations of motion of the system. The key point of the SINDy method is that many systems of interest are sparse and have the function $f_\theta(x(t))$ consisting of only a few terms from the space of possible functions. We will describe below how to set up the algorithm that learns $f_\theta(x(t))$ from data.

Consider gradient measurements at m time-steps for n states $y\epsilon \ \mathbb{R}^{m\times n}$, that are linear combinations of p functions from a feature matrix $\Theta(\mathbf{X}) \in \mathbb{R}^{m\times p}$ and entries of the coefficient vector $\xi \ \epsilon \ \mathbb{R}^p$ for each state. Performing a standard linear regression over ξ, the coefficients' vector, gives us each element's non-zero coefficients and bases contributions.

However, by adding a regularization term, λ, we can remove terms below a threshold for sparsity, resulting in the expression:

$$\xi = \underset{\xi'}{\operatorname{argmin}} ||\Theta\xi' - y||_2 + \lambda||\xi'||_1$$

The setup above is convex [3] and scales well to large-scale problems compared to brute-force combinatorial attempts. To determine the terms in the function f_θ from data, a time series of the general state $x(t)$ and its time derivative $\dot{x}(t)$ are sampled. This collected data is then stored in two matrices, one for the states and the other for their gradients, both of m time-steps for n states, as shown below.

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

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

Next we build a dictionary matrix of basis $\Theta(\mathbf{X}) \in \mathbb{R}^{m \times p}$ consisting of p candidate nonlinear functions for the m states of X as shown below:

$$\Theta(\mathbf{X}) = \begin{bmatrix} | & | & | & | & & | & | & \\ \mathbf{1} & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \dots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \dots \\ | & | & | & | & & | & | & \end{bmatrix}$$

The Lorenz system, for example, shown in the Results section later, has very few terms from the space of polynomial functions. In the $\Theta(\mathbf{X})$ matrix above, if we take $\mathbf{X}^{P_2}$ as an example, it would contain all combinations of polynomial state terms such that the sum of their powers would be quadratic, which can be a considerable subset depending on how many states we are considering [5]. As shown below

$$\mathbf{X}^{P_2} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1)x_2(t_1) & \dots & x_2^2(t_1) & \dots & x_n^2(t_1) \\ x_1^2(t_2) & x_1(t_2)x_2(t_2) & \dots & x_2^2(t_2) & \dots & x_n^2(t_2) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_1^2(t_m) & x_1(t_m)x_2(t_m) & \dots & x_2^2(t_m) & \dots & x_n^2(t_m) \end{bmatrix}$$

Each element of $\Theta(\mathbf{X})$ is a candidate for each element of the right-hand side, $f_\theta(x)$, of Equation 2.18. A linear regression problem can be set up to find the sparse matrix of coefficients $\Xi = [\xi_1, \xi_2 \quad ... \quad \xi_n]$ because only a few of the nonlinear functions are active in each column of f_θ, which represent the equation for each state in $\dot{\mathbf{X}}$.

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi \tag{2.19}$$

Each column ξ_k of Ξ is a sparse vector of coefficients that can tell us which terms are used in the right-hand side of each row of the equation $\dot{x}_k = f_k(x)$. Once we find Ξ, a model of each row of the governing equations can be built

$$\dot{x}_k = f_k(x) = \Xi^T(\Theta(x^T))^T \tag{2.20}$$

Thus, if we solve for the sparse vector of coefficients ξ for each state, we can find the dynamics of the equations [5]. Now that we have explained the method, the last thing to describe is what can be done if the measured data does not have the correct coordinates to predict true dynamic equations.

2.3 Intrinsic Coordinate Identification

A coordinate transformation is often necessary in dynamic system discovery to get the system's actual dynamics. A simple example is the charged particle Hamiltonian, e.g., for an electron. While the measured quantities of such a system are usually position and velocity, the Hamiltonian is expressed in terms of position and momentum. In a magnetic field, the conjugate momentum is not just *p=mv* but it is given by ***p = mv + eA(q)***, where *e* is the unit charge and *A* is the vector potential [40].

Fundamentally, SINDy relies on taking measurements in an effective coordinate system where the dynamics have a sparse representation [6]. However, measurements may only be accessible in a space where the dynamics are not sparse, so transforming to a different space is sometimes necessary to find the sparse dynamics. One of the most popular coordinate discovery methods is principal component analysis (PCA). It represents high-dimensional data in a low-dimensional linear subspace [6]. A nonlinear extension of PCA based on neural networks is the autoencoder.

Autoencoder based neural networks can be used to identify the intrinsic coordinates of a system. The intrinsic space can be of the same dimension as the measurement space or smaller dimensions. An autoencoder is a feed-forward neural network. Its innermost hidden layer represents the intrinsic coordinates that we are searching for. The network weights are trained to find an approximate input reconstruction for a certain number of intrinsic coordinates. Restrictions on the network can be of the type, number, and size of the hidden layers [15].

On their own, PCA and autoencoders do not take dynamics into account. The hallmark cases where neural networks shine through are computer vision and speech recognition, which are interpolatory. Forecasting is an extrapolation property. We seek dynamic

equations, where neural networks trained on historical data under-perform, especially for chaotic systems [6]. An additional problem faced by deep learning is the need for interpretability of resulting models. Attempts have been made at interpretation, but the sheer number of parameters is a hindrance [6]. Thus, the network must provide additional information to retain interpretability and retrieve extrapolatory models.

To circumvent this problem, we compose SINDy with an auto-encoder neural network, amalgamating the parsimony and interpretability of SINDy with the universal approximation abilities of deep neural networks to produce models useful for prediction. One can see an equation for neural network SINDy below:

$$\frac{d}{dt}z(t) = g(z(t)) \tag{2.21}$$

Where, the measurement coordinates are $\mathbf{x} \in \mathbb{R}^n$, and the reduced, learned coordinates are $z(t) = \varphi(x(t)) \in \mathbb{R}^d$, where $d \ll n$. g is the dynamic model containing only a few terms resulting from the transformation of coordinates x using the function φ, the encoder. Returning to the original coordinates is possible by mapping through $x \approx \psi(z)$, the decoder [6]. A sketch of the method can be found in Figure 2.1

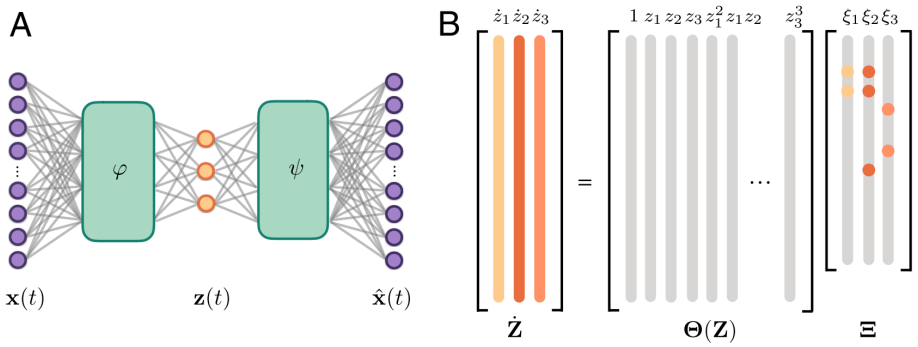


Figure 2.1: Neural SINDy Architecture and SINDy Matrices [6]

For better visualization, Figure 2.1 part A shows how the network's encoder-SINDy-decoder layers are composed. We transform coordinates from the measurement space to the intrinsic space, perform SINDy in the intrinsic space, and then transform back to the measurement space. In Figure 2.1 part B we see an example of the SINDy matrices setup, with the colored dots in the Ξ coefficients' matrix depicting the active coefficients after optimization.

Regular Physics informed neural networks (PINNs) employing autoencoder networks have been used for data-driven system identification, but there is no guarantee that the intrinsic coordinates will be sparse [6]. Therefore, constructing a neural network autoencoder with an encoder layer, then a SINDy layer, then a decoder layer is vital, where the SINDy layer works on sparsification, and the autoencoder part focuses on both encoding and reconstruction. Due to the sparsity and reconstruction requirements, adding terms to

the loss function that account for both constraints is essential. The loss function will have four terms, three loss terms, and one regularization term:

$$\mathcal{L} = \underbrace{\|\mathbf{x} - \psi(\mathbf{z})\|_2^2}_{\text{reconstruction loss}} + \underbrace{\lambda_1 \left\|(\nabla_{\mathbf{x}}\mathbf{z})\,\dot{\mathbf{x}} - \boldsymbol{\Theta}\left(\mathbf{z}^T\right)\boldsymbol{\Xi}\right\|_2^2}_{\text{SINDy loss in } \dot{\mathbf{z}}} + \underbrace{\lambda_2 \left\|\dot{\mathbf{x}} - (\nabla_{\mathbf{z}}\psi(\mathbf{z}))\left(\boldsymbol{\Theta}\left(\mathbf{z}^T\right)\boldsymbol{\Xi}\right)\right\|_2^2}_{\text{SINDy loss in } \dot{\mathbf{x}}} + \underbrace{\lambda_3\|\boldsymbol{\Xi}\|_1}_{\substack{\text{SINDy} \\ \text{regularization}}} . \quad (2.22)$$

The first term allows the autoencoder layers to act independently to convert back and forth from the intrinsic to the measurement space. It is crucial to have this term independent as it affects the functioning of the other loss terms. The second term is for SINDy loss in $\dot{z}$, i.e., in the intrinsic space. It uses a Jacobian of the encoded variables ($\frac{dz}{dx} = \nabla_x\varphi(x)$) and the measured gradients ($\dot{x}$) to find the gradient of the input data in the intrinsic space, $\dot{z} = \frac{dz}{dx}\dot{x} = \nabla_x\varphi(x(t))\dot{x}(t)$. This term is then compared against the predicted gradient of the encoded variables from SINDy, $\Theta(\varphi(x)^T)\Xi$. In the third term, the Jacobian of the encoded-decoded variables is taken with respect to the encoded variables, $\nabla_z\psi(\varphi(x)) = \frac{dx}{dz}$. This term is then multiplied with the predicted intrinsic space SINDy gradient $\Theta(\varphi(x)^T)\Xi$, which is a prediction for $\dot{x}$, comparing the network output with the true values. Finally, the last term is an L_1 regularization term on the coefficients that promotes sparsity. All the loss terms have associated hyper-parameters $\lambda_1, \lambda_2, \lambda_3$ that determine their weight relative to each other. These parameters can control the relative effects of these terms and allow more accurate prediction. Together, they comprise the loss function in Equation 2.22.

By constructing a proper loss function and network architecture, Champion et al. [6] show that it is possible to transform coordinates and find the dynamics of a Newtonian system in a reduced space. However, more is needed to guarantee it will also work for Hamiltonian dynamics.

From Chu and Hayashibe [8], we know that conducting SINDy model discovery on Lagrangian mechanics is possible. They carry out this task by holding the conservation of total energy as one of the constraints. By assuming the data on the changes in total energy with time, $\frac{dE}{dt}$, are available, they used singular value decomposition and least-square regression techniques to solve for the vector of dynamic equations coefficients. However, Hamiltonian mechanics behave differently.

Constraining the dynamics to hold a symplectic form imposes a more restrictive condition on the possible solutions. Using total energy as the constraint while searching for Hamiltonian dynamic equations will not guarantee that we obtain suitable basis functions, as the loss function is optimized. Since the symplectic structure, not the total energy, completely characterizes Hamiltonian systems [19], we should aim to have the symplectic structure as a constraint. Due to the nature of neural networks, it is more likely that basis equations that do not represent actual dynamics but still satisfy the total energy-dependent loss function are found. A similar occurrence was shown for Newtonian systems by Cham-

pion et al. [6]. Therefore, when setting up our Hamiltonian-SINDy problem later, we aim to maintain the symplectic structure instead of total energy as a tenet of the method.

In the following chapters, we will first show that Hamiltonian-SINDy works while preserving the symplectic structure. Then, we will show an example of a coordinate-transforming, auto-encoder neural network working on a nonlinear Hamiltonian system to show how coordinate transformation with Hamiltonian-SINDy can be generalized.

3 Methodology

This section will discuss the code, techniques, and test setups to reach a canonical-conjugate-coordinate and vector field discovering Hamiltonian-SINDy method.

We chose Julia for the implementation because it is widely used in the scientific community for its user-friendly syntax and high performance [25]. It is not subject to the two language problem like Python [36] and is known to perform fast linear algebra computations for various test cases.

First, we will implement the original SINDy algorithm [5] and reproduce results from the literature in the Julia programming language. After re-creating the original results, we will evaluate different optimization strategies to verify if it is possible to formulate the SINDY problem as an unconstrained optimization problem, as this will be required for Hamiltonian SINDy. Next, we will apply SINDy to Hamiltonian systems and test the code on various physics models. After that, a variant of SINDy will be implemented to discover the governing equations from just the information about the states at different time steps without using gradients, as the gradient information is often unavailable. In the next step, we will recreate a variation of classical SINDy, incorporating an autoencoder that can discover an intrinsic coordinate space to allow for a potentially sparser SINDy implementation.

Finally, we will attempt to assemble an Autoencoder-Hamiltonian-SINDy method, using it to reproduce results obtained from the Hamiltonian-SINDy method implemented before while simultaneously discovering the conjugate coordinates of the Hamiltonian. The sections below will describe the implementation of the algorithms, data structures, and some test cases used to implement and validate the methods. We will emphasize the steps necessary to adapt the classical SINDy code to the Autoencoder-Hamiltonian-SINDy method.

3.1 SINDy Algorithm

The SINDy algorithm is required to solve a regression problem for optimization. There are many choices for optimization algorithms available to solve for the matrix of sparse coefficients, Ξ, from Equation 2.19. Here, the traditional algorithm will be discussed, how problems are set up, the optimization techniques that can be used to solve them, and differences in the setup from the original implementation.

There is a distinct sparse coefficients' vector, ξ, for each of the states in Equation 2.19, and together they make up the sparse coefficients matrix, Ξ. As seen before, the matrix $\Theta(\mathbf{X})$

serves as a dictionary of p candidate basis functions, evaluated at m time-steps, making its dimensions $m \times p$. The time steps are taken to be many more than the candidate functions, $m \gg p$, to allow for higher accuracy and to avoid over-fitting to noisy data. As realistically most measured data is noise-contaminated, independent identically distributed Gaussian noise with zero mean is added to the gradient to simulate the noisy data. Thus, the gradient data will be assumed to be:

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi + \eta\mathbf{Z}, \tag{3.1}$$

where $\mathbf{Z}$ is the Gaussian noise matrix, and η is the noise magnitude. In the original paper [5], the sequential threshold, least-squares regression algorithm was used to solve equation 3.1. The least-squares regression method tries to transform the matrix A in the equation $Ax = b$ to reduce its size before the inverse is calculated. If one takes the matrix as $\Theta(\mathbf{X}) = A$ of size $m \times p$, then $A^T A$ is a smaller matrix of size $p \times p$. To simplify solving for $\Xi = x$, using A and $\dot{X} = b$, we can then write

$$Ax = b \rightarrow A^T Ax = A^T b \rightarrow x = (A^T A)^{-1} A^T b. \tag{3.2}$$

This equation is solved using least-squares regression when A is rectangular or by a linear solver when A is square. The least-squares algorithm is performed on basis values in Θ and the gradients $\dot{x}$, as seen in the SINDy Algorithm in Code 3.1. After getting an initial solution for Ξ, using the basis dictionary, coefficients smaller than some threshold value λ are set to zero, and the corresponding basis functions are removed from the dictionary. The threshold is applied repeatedly until no more coefficients with values below the threshold parameter exist. The advantage here is the algorithm's rapid convergence to a sparse solution and the simplicity of using one parameter λ for sparsification. This algorithm is robust to noise in the gradient data, as we will see.

In the algorithm 3.1, we supply the sparsification function with data for gradients, *xgrad*, and a dictionary of candidate basis functions Θ, add noise to the gradient data, and compute an initial guess for the coefficients of the basis functions by performing least-squares regression as in Equation 3.2. Then, we check if any coefficients are smaller than the threshold sparsification parameter and repeat least-squares regression for several consecutive loops. The loop breaks if no coefficients smaller than the threshold are found; otherwise, we set the smaller coefficients to zero. In the subsequent linear regressions, discarding all basis functions whose coefficients have been set to zero is essential. $\Theta[:, biginds]$ indicates, for all samples, which of the bases are still active for the selected state "***ind*** ." Only the corresponding noisy gradients of each state are then chosen to perform the linear regression "*xgrad_noisy[ind,:]*." Linear regression has to be performed over the system's states one at a time because different coefficients are active for different states. Since the active terms in each state's basis vectors are usually not the same, it prevents the action of the least-squares algorithm on the whole coefficients matrix at once.

The algorithm is efficient while providing high accuracy and robustness, even if the sizes of the matrices are large. One can see how resistant the algorithm is to noise in Figure 3.1.

SINDy Algorithm

```
# add noise
xgrad_noisy = xgrad .+ method.noise_level .* randn(size(xgrad))

# initial guess: least-squares
Ξ = Θ \ xgrad_noisy'

for _ in 1:method.nloops
    # find coefficients below lambda threshold
    smallinds = abs.(Ξ) .< method.lambda

    # check if there are any small coefficients != 0 left
    all(Ξ[smallinds] .== 0) && break

    # Set all small coefficients to zero
    Ξ[smallinds] .= 0

    # Regress dynamics onto remaining terms to find sparse Ξ
    for ind in axes(xgrad_noisy,1)
        biginds = .~(smallinds[:,ind])

        Ξ[biginds,ind] .= Θ[:,biginds]\xgrad_noisy[ind,:]
    end
end
```

Source Code 3.1: SINDy Algorithm: Uses the backslash operator, which uses either a linear solver or least-squares regression, depending on the system it is applied to, i.e., a square or rectangular matrix

We used uniformly sampled data in the range [-20, 20], a default sparsification parameter of 0.05, and just 144 samples for a two-dimensional damped harmonic linear oscillator system. Then we added random Gaussian noise with a maximum amplitude of 10% of the sampled range. The algorithm takes just 0.001788 seconds to run, and in one SINDy cycle, it outputs the results in Figure 3.1. The original code samples one trajectory over a specific time period and finds the gradients at those samples with the theoretical reference equation. We sample uniformly in a range and take gradients. It is faster because we do not calculate a trajectory, and the sampling can be from a broad controlled sample area, which can be customized if needed. Despite the high noise added, the discovered coefficient values, selected from 21 possible polynomial basis functions up to the 5^{th} power for each state, are the correct basis functions with only one extra constant bias coefficient showing up, as can be seen in Figure 1 in the Appendix. All the correct discovered basis functions are accurate within one decimal place of their actual value. From plot 3.1, for the dynamics of the linear oscillator over a relatively long time period of 25 time units, it is clear that the SINDy algorithm is highly resistant to noise as it manages to maintain the correct trajectory quite closely.

The Least squares regression algorithm is reliable but only works for linear relationships between variables. To allow more flexibility in the variable relationships we can handle, we test if the SINDy method works with an unconstrained optimization setup in the next step.

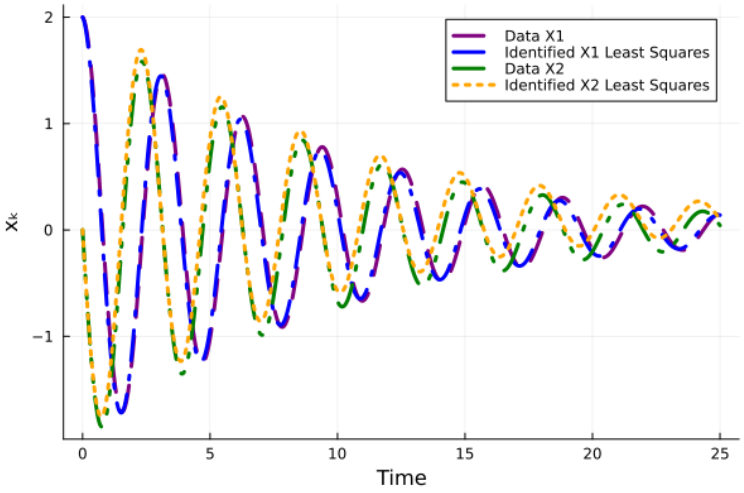


Figure 3.1: SINDy State Plots with high noise

3.1.1 SINDy with Unconstrained optimization

Least-squares regression can only be employed when the relationship between variables is linear. However, in our Hamiltonian-SINDy and Autoencoder-Hamiltonian-SINDy methods, it would be problematic to use the Least-squares algorithm because the relationship between variables in these methods is usually nonlinear. This chapter discusses alternative optimizers in the context of classical SINDy, with the goal being to change from the Least-squares optimization technique to algorithms like Broyden–Fletcher–Goldfarb–Shanno (BFGS) [13], [39], Conjugate Gradient (CG), Gradient Descent (GD) or Newton's Method [35] which can handle a broader class of problems. We will choose a Julia package to call these solvers.

Before optimizing, we need to construct an appropriate loss function that measures the error between the model's prediction and the desired outcome and has to be minimized [35]. We use the squared L^2 norm of the differences of the predicted and reference vector fields:

$$\sum_{i=1}^{m} (\dot{X}_i - \Theta_i(X)\Xi)^2. \tag{3.3}$$

For optimization, we employ the Optim.jl package [31], which implements a large number of algorithms. An ideal algorithm combines high accuracy with low memory footprint and computational cost. From the popular algorithms available on Optim.jl [31], theoretically, Newton has the highest accuracy and cost due to its requirement of the second derivative of the loss function. However, it is sensitive to the initial guess of the coefficients [35]. Moving to the Quasi-Newton methods like Gradient Descent (GD), BFGS, L-BFGS, and the Conjugate Gradient method (CG), these are known to perform well while

using an approximation of the objective loss function's hessian or simply its gradient for calculations. We also want the algorithm to be robust to different initial guesses of the coefficients. To test the algorithms for accuracy, we will compare the trajectories from the reference gradients to the ones found by the SINDy optimizers for 25 time units and make a sum of errors versus the percentage input noise plot. We set up our linear oscillator system example from this criteria and plot errors versus noise amplitude, as shown in Figure 3.2. The figure shows that BFGS, with coefficients initialized to zero, has the lowest error at the highest noise input, with most other setups also performing comparatively well. Table 3.1 below compares optimization metrics between these different algorithms.

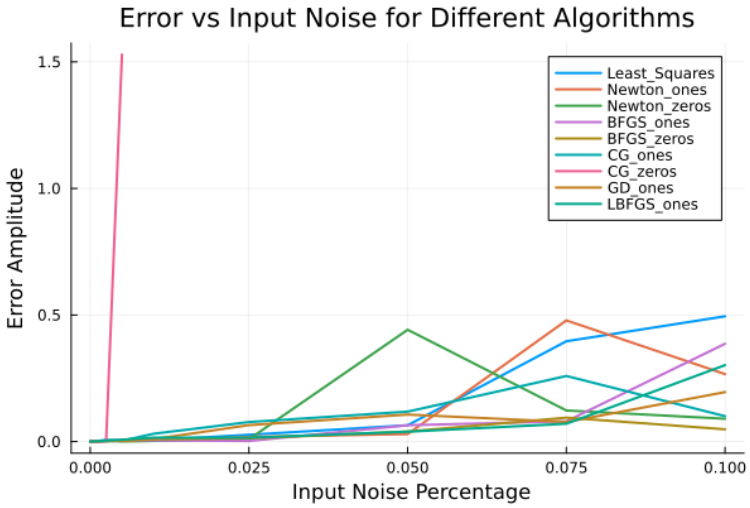


Figure 3.2: error Vs. Noise Across Algorithms and Initializations in a Linear Oscillator

Method	Computation Time (seconds)	Iterations	SINDy cycles	Coefficients Residual
Least-Squares	0.0024	-	1	0.0003
BFGS-zeros	0.17	65	1	0.0003
BFGS-ones	2.138	507	1	0.0001
Newton-zeros	4.376	105	1	0.0004
Newton-ones	19.296	504	1	0.0003
CG-ones	1.610	514	2	0.0004
LBFGS-ones	2.885	523	2	0.0001
GD-ones	2.658	524	4	0.0002

Table 3.1: Performance Metrics for Optimization Methods with Linear Oscillator

In Table 3.1, "Coefficients Residual" is the maximum difference between the predicted and reference vector field coefficients' values, which is different from the error amplitude calculations in Figure 3.2 that implicitly incorporate all the coefficients into the error calculation. The data is sampled at a noise amplitude of 0.1, with 400 samples, and maximum iterations set to 500 per optimizer call to limit runtime. The measurements show that the

BFGS and **LBFGS** algorithms, with all SINDy coefficients initialized to ones, have the highest accuracy regarding the maximum coefficient's residual. BFGS-zero records the shortest computation time. However, all the algorithms have high accuracy and are comparable, as seen from Figure 3.2 and Table 3.1. Most of the algorithms take many iterations to converge, which also adds to their run time. However, limiting iterations and performing more SINDy cycles does not produce a worse final coefficients' residual in this example. All algorithms converged within two SINDy loops, except Gradient Descent (GD), which might be due to the method's reliance on just the gradient information and not the hessian. Relying solely on the gradient could also lead to long runtimes if the problem is ill-conditioned [31]. Most of the algorithms outperform Least-Squares regression in terms of accuracy for high noise input, as seen from Figure 3.2. We will choose BFGS initialized to zeros as our optimizer and initialization of choice due to its good accuracy, short runtime, and few iterations. However, the main reason for not choosing the other algorithms is that most could not provide usable results for high noise inputs with coefficients initialized to zero. Other than BFGS and Newton, only the Conjugate Gradient could provide some results with zero initialization but only up to small noise inputs.

An initialization of coefficients set to zero should be able to sparsify quickly and execute faster because most unnecessary coefficients remain close to zero, and it takes fewer loops to remove them. This is also seen from the computation time in Table 3.1.

Lastly, we note that with initialization of all coefficients equal to one, BFGS provides the worst accuracy with respect to high noise, as seen from Figure 3.2. However, it is still comparable to the accuracy provided by the other algorithms. It also still outputs a very low maximum coefficients' residual. From these findings, we focused on the **BFGS** algorithm for subsequent experiments with an initialization of coefficients equal to zero. Newton with zero initial guesses will be used when a higher order method is required, although this comes with a higher computation time because it needs to calculate the loss function's gradient and hessian. An initial guess of ones can also be used later with different algorithms to further study robustness with respect to the initial guess.

Now that we have verified that SINDy can perform well with optimization algorithms other than Least-Squares, it is time to adapt the SINDy approach towards symplectic structure-preserving Hamiltonian systems. This method will be described in the next section.

3.2 Hamiltonian-SINDy Algorithm

When constructing the Hamiltonian SINDy method, the coefficients cannot be as easily factorized into a linear system of equations for the coefficients as in traditional SINDy. In traditional SINDy, the vector field can be written as a matrix-vector product of coefficients and basis functions in a very immediate way. In Hamiltonian SINDy the vector field is more complicated as it arises from the gradient of a parameterized Hamiltonian, and expressing it as a matrix-vector product is not as immediate. Still, the vector field usu-

ally depends linearly on the coefficients and thus can, in principle, be transformed into a matrix-vector product. It is just much more involved to do this.

This section will explain how to generate a dictionary of Hamiltonian vector field functions with a symplectic structure for SINDy to optimize. We want the candidate basis functions to comprise typical functions used for approximation: polynomial, trigonometric, exponential, logarithmic, and combinations thereof. Moreover, we want the vector field to maintain its skew-symmetric symplectic structure throughout the optimization. Subsequently, we would like the SINDy algorithm to be able to function while only having data for states at two time steps and no gradient data available. Considering these criteria, we will discuss how the setup was implemented and the problems encountered.

Let us start by assuming we prepossess an array of basis functions; how should we then acquire the skew-symmetric Hamiltonian vector field? First, we need to find the number of coefficients for the basis functions. If there is no repetition in the bases, the number of coefficients should correspond to the number of basis functions. We then construct an array of coefficients of the same size, multiply each coefficient with a basis function, collect the results, and sum them. These steps will give us a Hamiltonian of all the terms from the basis dictionary.

To support the creation and testing of the Hamiltonian vector field, we use symbolic math for the new basis assembling technique to convert the symbolic expressions to a function later. Symbolic math, or algebraic manipulation, involves performing algebraic operations on symbolic expressions, where the variables and quantities are represented as symbols rather than numbers held within variables. We use the *Symbolics.jl* package for this, since it contains differentiation, simplification, function generation, and much more in its arsenal [16]. It is beneficial for prototyping and setup, as the results of the calculations are readable instead of just outputted numbers, allowing analysis and verification.

Now, we need to differentiate the Hamiltonian with respect to the elements of Array 3.4 i.e., $z = q_1, ..., p_2$. We can use a function of the *Symbolics.jl* package for differentiation, obtaining the derivative of the Hamiltonian. Afterward, we multiply the derivatives with a skew-symmetric matrix conforming to the symplectic structure of Equation 2.15 to get the correct Hamiltonian vector field. Finally, we use the *RuntimeGeneratedFunctions.jl* package to generate and return a callable Julia function that returns the Hamiltonian vector field. This generated function takes as arguments a storage variable for the gradients, the system's current state z on which to calculate the gradients, and the Hamiltonian coefficients to return a vector field with built-in symplectic structure. The steps described above can be seen in Code 3.2:

Now that we know how to construct the Hamiltonian vector field from the basis, we will attempt to construct functions to generate symbolic basis functions. Starting with a simple example, we define a symbolic variable array holding four symbolic variables to represent

Hamiltonian Vector-Field Function Generator

```julia
using Symbolics
# returns a function that can build the gradient of the Hamiltonian
function ΔH_func_builder(d::Int, z::Vector{Symbolics.Num} = get_z_vector(d),
    basis::Vector{Symbolics.Num}...)
    # nd is the total number of dimensions of all the states
    nd = 2d
    Dz = Differential.(z)
    # collects and sums combinations of basis and coefficients."
    basis = get_basis_set(basis...)
    # gets the number of terms in the basis
    @variables a[1:get_numCoeffs(basis)]
    # collect and sum combinations of basis and coefficients
    ham = sum(collect(a .* basis))
    # gives the derivative of the Hamiltonian but not the skew-symmetric true one
    f = [expand_derivatives(dz(ham)) for dz in Dz]
    #simplify the expression potentially to make it faster
    f = simplify(f)
    # multiplying with the skew-symmetric matrix
    ΔH = vcat(f[d+1:2d], -f[1:d])
    # Calculatesgradientt and converts to a native Julia function
    ΔH_eval = @RuntimeGeneratedFunction(Symbolics.inject_registered
        _module_functions(build_function(ΔH, z, a)[2]))
    return ΔH_eval
end

```

Source Code 3.2: Hamiltonian Vector-Field Function Generator

a 2-dimensional system:

$$\begin{bmatrix} q_1 \\ q_2 \\ p_1 \\ p_2 \end{bmatrix} \tag{3.4}$$

The goal is to set up simple functions that provide an easy interface for the user to get many different types of symbolic bases of arbitrary dimensions in an array, which can then be presented as an argument to the Hamiltonian vector field function generator from Code 3.2.

The original Matlab implementation of the SINDy algorithm only considered polynomial basis functions up to degree 5 and trigonometric functions with mode numbers up to 10. In our Julia implementation, we support a larger range of basis functions, including polynomials of arbitrary degree, trigonometric functions with arbitrary mode numbers, as well as rational, exponential and logarithmic functions. All of these basis functions can be applied to the state variables' individual components or functions thereof, such as arithmetic combinations. As the basis dictionaries are first created symbolically, they can be printed in an easily readable form, allowing quick debugging.

Some functions are indeed just identities of others, which is valid for some trigonometric, exponential, and logarithmic functions. However, we want to provide maximum flexibility to potential users while expecting them to know which functions in the generated basis dictionary are identical and exclude those themselves. The result of this endeavor can be found in our GitHub Repository 1. An example of the range of Hamiltonian functions

created just from the variables in Array 3.4 can be seen in Code 3.3. Using code 3.3 and parts of code 3.2 without the function generator line, we can see what a potential symbolic skew-symmetric Hamiltonian vector field would look like in Figure 3.3.

```
# dimensions of Hamiltonian variables (q,p)
d = 2
# symbolic array of variables (2D)
z = get_z_vector(d)
# basis functions
trigonometric  = trigonometric_basis(z, operator = /, max_coeff=3)
logarithmic = logarithmic_basis(z, polyorder = 1, operator = -)
exponential_diff  = exponential_basis(z, polyorder=1)
mixed_basis = mixed_states_basis(logarithmic, trigonometric)
mixed_basis = mixed_states_basis(mixed_basis, exponential_diff)
basis = get_basis_set(mixed_basis)
# coeffcients: (a)
@variables a[1:get_numCoeffs(basis[3013:3015])]
# collect and sum combinations of basis and coefficients
ham = sum(collect(a .* basis[3013:3015]))
Dz = Differential.(z)
# derivative of the Hamiltonian, but not skew-symmetric
f = [expand_derivatives(dz(ham)) for dz in Dz]
f = simplify(f)
# multiplying with the skew-symmetric matrix
ΔH = vcat(f[d+1:2d], - f[1:d])
```

Source Code 3.3: Symbolic skew-symmetric Hamiltonian vector field code example

$$\begin{bmatrix} 2a_{15}e^{2(-p_2+q_1)} + 3a_{27}e^{3(-p_2+q_1)} + a_3e^{-p_2+q_1} + a_{70}\cos(-p_2+q_1) - a_{82}\sin(-p_2+q_1) \\ 2a_{16}e^{2(-p_1+q_2)} + 3a_{28}e^{3(-p_1+q_2)} + a_4e^{-p_1+q_2} + a_{71}\cos(-p_1+q_2) - a_{83}\sin(-p_1+q_2) \\ -a_1e^{q_1-q_2} - 2a_{13}e^{2(q_1-q_2)} - 3a_{25}e^{3(q_1-q_2)} - a_{68}\cos(q_1-q_2) + a_{80}\sin(q_1-q_2) \\ -2a_{14}e^{2(-p_1+q_1)} - a_2e^{-p_1+q_1} - 3a_{26}e^{3(-p_1+q_1)} - a_{69}\cos(-p_1+q_1) + a_{81}\sin(-p_1+q_1) \end{bmatrix}$$

Figure 3.3: Skew-Symmetric Hamiltonian SINDy Gradient

So far, setting up the basis and getting a function that outputs its Hamiltonian vector field has been accomplished. Now, we write a sparsification function to work with these equations. A problem arises that once coefficients have been removed in the sparsification, they should be excluded from the optimization problem in the next sparsification cycle. However, the whole coefficients' vector must be input to the vector field generating function that calculates the predicted gradient. Lets call the function `hamsindy_grad()`. To ensure the discarded coefficients are ignored, we create a bit-vector called *biginds*, of the size of the original coefficients vector. This bit vector is true at elements where the coefficients are greater than the sparsification threshold (λ) and false when coefficients are less than or equal. During optimization, we create a smaller vector by placing the bit-matrix at the indices of the coefficients greater than λ, as *big_coeffs=coeffs[biginds]*. Then we pass the vector *big_coeffs* to a wrapper loss function, called `sparse_loss(big_coeffs)`. This way, only the active coefficients will be optimized. Inside `sparse_loss`, we define a zero vector of the size of the original coefficients vector and set *big_coeffs* to the indices of this

new vector corresponding to *biginds*, as *new_coeffs[biginds]=big_coeffs* and pass this vector to the actual loss function, *loss(new_coeffs)*. In this way, only the active, large coefficients are optimized each time, and the rest are ignored, as required. With this, we finish setting up our Hamiltonian-SINDy method, the code for which can be found in our GitHub Repository 1.

Appropriate models were then considered to test the algorithm. To cover a variety of cases with different properties, we test on examples including a nonlinear oscillator, the Toda lattice, point-vortices, and a solar system model. These examples incorporate a wide array of relevant nonlinear models and basis function types, which show the accuracy and limitations of the technique developed here. The results from these experiments will be discussed in section 4.2.

Our next step is to modify the Hamiltonian-SINDy algorithm to operate on just state data. The Hamiltonian vector field data is often challenging to estimate, noisy, or unavailable. For cases like these, it would be conducive if the algorithm could function without needing reference gradient data.

3.2.1 Hamiltonian-SINDy Without Gradient Data

Developing a method that relies on something other than reference gradient data allows us to start with even less information that we need to provide the method. Although this is an engaging test for the algorithm's robustness, there are also practical reasons why the method needs to function well enough without vector field data. For example, in complex many-body systems, the Hamiltonian vector field data is very expensive to compute if all constituents interact. In other examples, when the SINDy algorithm is applied to identify dynamical systems from experimental data, vector field data may not be attainable.

We attempt here to get results from the algorithm by only supplying it with state data at two time points, with a short interval in between, along with a way to interpolate and estimate the gradient. The technique will then be checked for accuracy and speed against the original Hamiltonian SINDy algorithm that requires gradient data. We will keep the implementation simple to have only a proof of concept available for future improvement. Below, we will discuss implementing such a gradient-free basis discovery technique.

The idea is to replace the L^2 norm of the difference in the vector field with some numerical integrator. Noisy experimental data is simulated by taking state values separated by a short time interval and introducing a noise factor in the second state result. We use an implicit Runge-Kutta geometric integrator function *Gauss(4)* from the GeometricIntegrators.jl package [27] to obtain high-accuracy training data for the state at the second time point and add noise to it. Modifying the sparsification function, we start by finding the SINDy-gradient of the state at the first time point, which is our initial guess. Next, we need to find the solution at the next time point by integration. Within each optimization step, the implicit midpoint method is solved using Picard's method [38] with a fixed number of iterations. After iteratively improving the guess for the second state, the SINDy prediction is compared to the reference noisy state solution at the second time point. Then, an opti-

mizer such as Newton or BFGS is used to minimize the difference between the prediction and reference. This way, the training of the method can be performed without the need for gradient data.

We show comparative results through the nonlinear oscillator example from Section 4.2.1. The method is kept simple by using Explicit Euler instead of a higher-order method like Runge-Kutta method, as the time-step between states is small. The calculation still gives accurate results up to 2 decimal places for the nonlinear oscillator toy model, even with some noise added to the input, which is 2.5% of the maximum sample range. Upon starting the SINDy procedure with an initial dictionary of 97 basis functions, the algorithm identifies four basis functions as sufficient to describe the data within a tolerance of the coefficient's residual of 0.05. This shows that the algorithm is robust even when provided little information about the system (in the form of a more specialized set of basis functions). BFGS was used as the optimizer with 1000 sample states taken uniformly from the range (-20,20). The computation time, loss function residual, and other relevant metrics from running this experiment for both Hamiltonian SINDy methods can be seen in Table 3.2.

Method	Computation Time (sec)	Iterations	Loss Function Residual
Ham-SINDy	16	83	4.011323e+02
Ham-SINDy no gradient	1124	115	4.081489e+02

Table 3.2: Example Performance Metrics for Hamiltonian SINDy Methods

The loss function residual of the Hamiltonian SINDy method using only state data is comparable to that of the Hamiltonian SINDy method using vector field data, as seen in Table 3.2. The "Loss Function Residual" records the sum of the loss function values from all data samples. Ultimately, we get accurate coefficients up to 2 decimal places from this alternative setup. In contrast, the normal Hamiltonian SINDy gives accurate coefficients to 5 decimal places. It can be seen in Table 3.2 that it takes much more computation time for the method using Picard iterations to reach a similar level of accuracy. This slowdown is expected due to the repeated gradient calculations inside the Picard iteration loop and the extra effort required from the optimizer while calculating the gradient of the loss kernel, all of which add to the computational effort. The difference in the accuracy of the resultant coefficients between the version that uses and does not use gradient data reduces when the noise is removed. This increase in accuracy tells us that the noise introduced plays some role in the convergence of the loss function in the gradient-less version. Even though we observe that the errors in the coefficients grow with increasing noise levels, we find that the SINDy algorithm determines the correct constituents of the Hamiltonian. We also see that despite running for more iterations, the Picard iteration version cannot reach the same accuracy, which indicates that the noise introduced and the explicit Euler integrator limit accuracy. Trying a higher-order integration method in the Picard iteration, more Picard loop iterations, or just introducing less noise could improve the accuracy but would probably also increase the runtime of the solver.

From the information gathered, the state data employing version of Hamiltonian SINDy

provides a suitable alternative if the gradient information is unavailable. The Hamiltonian-SINDy method has now been set up to work both with and without the gradient, and its performance has been tested to be sufficiently accurate and fast. It is time to move to another implementation that tackles the coordinate transformation problem while simultaneously performing SINDy.

3.3 Intrinsic Coordinate Identification with SINDy

Classical SINDy relies entirely on the collected data from an effective coordinate system. However, data can only sometimes be collected in an effective coordinated system. Experimentally accessible observables might not provide the most compact representation of a system. Moreover, there can be situations where the system is represented with sufficient accuracy in a reduced space where calculations are easier to perform. In this case, reducing to a smaller state space saves computational effort while retaining predictive accuracy. Even more important is that the measured data for Hamiltonian systems is often not in the canonical-conjugate coordinates. Converting to these coordinates gives us access to many interesting features of the Hamiltonian formulation, such as preserving the symplectic structure during canonical transformations that potentially make the Hamiltonian easier to solve [41]. Therefore, it is convenient for Hamiltonian systems to have data in canonical-conjugate coordinates.

In this section, we will couple an autoencoder with SINDy in a "Neural-SINDy" method, in the Julia language while discussing important implementation choices that differ from the Champion et al. [6] paper where this method was first presented.

The size of the systems we can identify using SINDy is restricted by the computational resources available. We will mention the changes made to the original network from Champion et al. [6] to allow the simulation to run in a feasible time on a typical development system. We limit the encoder and decoder sizes to work without hidden layers and without using activation functions. There will be just one intrinsic layer, and the layer's size will be the same as the input. In summary, we use a randomized linear transformation on the data. This is to be understood as a first proof-of-principle of intrinsic coordinate identification within the framework of Hamiltonian SINDy. Fully developing this approach goes beyond this thesis project's scope but certainly warrants further research.

We will limit the training epochs to 2000 for the initial optimization and 500 for each SINDy cycle and the final optimization. In contrast, the original paper had 5000 epochs initially and, subsequently, 1000 epochs for the SINDy and final optimizations. We chose the Lorenz attractor [30] example from Champion et al. [6] and aim to re-discover its governing equations using the current method with sufficient accuracy. Instead of going from a space of 128 states to 3 states, to keep the example simple, the number of states will be kept constant, starting from 3 states and having 3 states in the intrinsic layer. All network parameters will be initialized using Xavier initialization [14], whereas SINDy coefficients will be initialized to zeros.

The Lorenz example from Champion et al. [6] was trained with 1024 trajectories with 250 points for each trajectory, counting to 256000 sample data points. In contrast, we will use a uniform sampling from the range (-20,20), picking 3375 samples. In this initial setup, the Julia package Flux.jl [22] will be used to create the model architecture with the Zygote [21] backend for automatic differentiation. Flux is a popular machine learning package in Julia known for its simplicity and extendability [22]. We chose it here to have an easy way to apply SINDy in conjunction with an encoder-decoder pair.

The strategy employed will be that a tuple of a model with three parametric layers, i.e., encoder layer, SINDy layer, and decoder layer, will be built. This model will pass the data along with a dictionary of 56 basis functions for the Lorenz example to a solver function. This function will divide the data into batches and shuffle the data points in a batch for each epoch. This technique is called stochastic mini-batching. It helps the model learn from different patterns in the system. We will experiment with different batch sizes to compare convergence against noise. For each batch, the quantities required by the loss function, such as the gradients of the model layers $\frac{dz}{dx} = \frac{d\varphi(x)}{dx}$ and $\frac{dx}{dz} = \frac{d\psi(\varphi(x))}{dx}$ as explained in Equation 2.22, will be calculated.

As described in [6], the hyperparameters λ_1 and λ_2 as they appear in Equation 2.22 will be slightly smaller than the ratio between the states and the gradients, divided by specific amplitudes for relative weighting. This ratio ensures that the auto-encoder layers focus more on reproducing the states, while the SINDy-layer focuses more on the gradient prediction [6]. The regularization parameter λ_3 balances sparsity, overfitting, and prediction and is the one that typically requires the most tuning. This last parameter λ_3, along with different batch sizes, will be focused on while attempting to improve accuracy by exploring a range of possible values, a common hyperparameter tuning technique.

Lastly, same as the original paper, the Adam optimizer [26] with default learning rate and momentum will be used to update the states. It is important to recall that because Adam uses the previous gradient updates for its momentum calculations, defining the Flux optimization setup outside any loops is crucial. In our algorithm, we chose to redefine the Adam optimization state at each SINDy cycle because the coefficients left after sparsification compromise a new system. Therefore, the remaining coefficients should not be affected by gradient calculations from the previous optimizations. This setup effectively removes both the effects on the Adam optimization state from previous iterations and the coefficients below the sparsification threshold. This step is also what is done by Champion et al. [6], and the result can be seen in Figure 3.4.

Since Zygote is cumbersome to set up, restricting certain functionalities like composing a gradient function on a jacobian function, which we require, some syntactic workarounds had to be developed around these limitations. Due to these problems, the *Enzyme* [32] and *ForwardDiff* [37] packages were also tested as possible candidates for automatic differentiation rather than Zygote. However, Enzyme proved unfeasible for this study as some features were incompatible.

The *ForwardDiff* package was also tested as a replacement for Zygote, and initial proto-

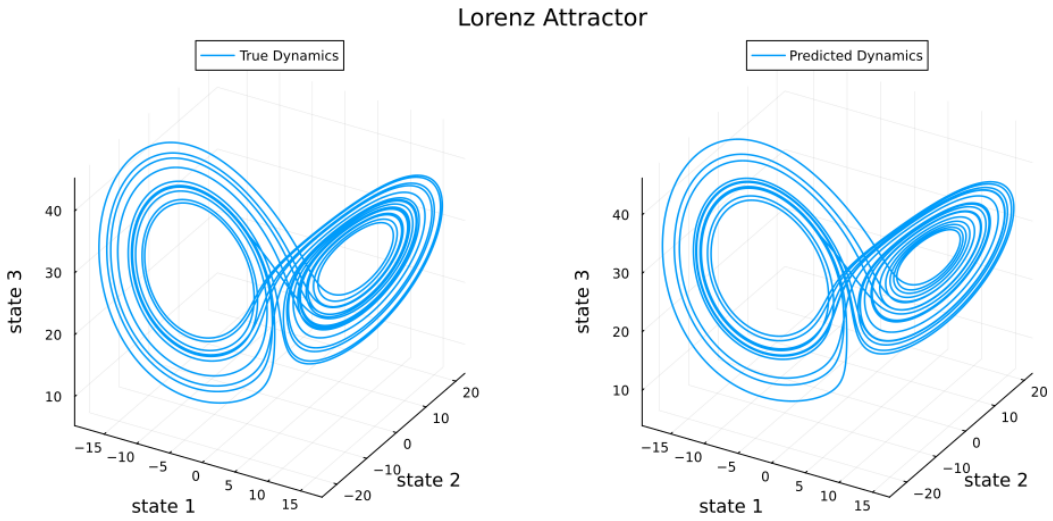


Figure 3.4: Neural SINDy Prediction versus True Dynamics for the Lorenz System

typing was carried out with it. However, ForwardDiff requires explicitly flattening models before passing them for gradient and jacobian calculations. Moreover, it did not seem to provide syntactic or computational advantages over Zygote during setup. It was also observed to have disadvantages when eventually moving to Hamiltonian-SINDy due to the complexity of the *RuntimeGeneratedFunction* for the SINDy vector field and the need to differentiate through it during optimization.

Therefore, it proved more straightforward to use Zygote and work around its limitations with mutating arrays. In the end, the setup for the algorithm was similar to the classical SINDy one, where an initial encoding was carried out, followed by a SINDy optimization and finally a decoding step to reconvert to the measurement state variables. Three optimizations are carried out with these steps according to the Champion et al. [6] algorithm. The first is longer with 2000 epochs, an initial optimization without sparsification similar to the step in classical SINDy. Then, a sparsification loop and an additional last optimization are made for 500 epochs each to allow the parameters and loss residual to converge further.

The results from the Lorenz attractor prediction are shown in Figure 3.4 with 18 coefficients selected by the algorithm from a basis with 5^{th} order polynomial combinations of 56 candidate basis functions. The original paper [6] had five models with 15 or more active terms from the ten models that they tested. Figure 3.4 shows that the prediction reproduces the Lorenz attractor dynamics within sufficiently small error bounds. This example setup will be discussed in more detail in Chapter 4.3.2. Although the method identifies a vector field with more terms than the true dynamics, it still gives seemingly good results for trajectory prediction, which is the main goal of the Neural-SINDy method. Now, we

can finally move to setting up an Auto-encoder-Hamiltonian-SINDy algorithm.

3.4 Auto-Encoder-Hamiltonian-SINDy Algorithm

The algorithm presented in this section is quite similar to intrinsic coordinate identification setup from section 3.3. The main idea is to replace the classical SINDy layer with a Hamiltonian-SINDy layer. However, this imposes intricate challenges. As seen from the loss function in Equation 2.22, there are inter-dependencies between the encoder-decoder weights and the coefficients of the SINDy parametrization. Due to the universal approximation capabilities of neural networks, it is possible to find encoder-decoder weights that do not convert the data to canonical conjugate coordinates but can encode and decode the data with high accuracy. SINDy then predicts gradients on this wrongly encoded data, identifying incorrect dictionary basis functions. Yet, the loss function from Equation 2.22 decreases and converges to a sufficiently small value. Unlike in classical SINDy, in Hamiltonian-SINDy, it is crucial to have the correct coordinates in intrinsic space; otherwise, neither the long-time dynamics nor the vector field equations will be correct. This problem was not faced by Champion et al. [6] using classical SINDy as the additional dictionary basis functions that were identified could still produce the trajectory plots with sufficient accuracy, and their goal was focused on producing correct trajectories rather than finding the exact governing equations.

Moreover, since the Hamiltonian gradient vector contains elements consisting of the sums of candidate basis functions and their coefficients, it is much more difficult to find correct coefficients in Hamiltonian-SINDy than in classical SINDy, where the vector field has a simpler structure. The autoencoder exacerbates this problem, and larger dictionary sizes add to this problem. It was also noticed during testing that the differences in the size of the variables' data values, for example, having large momentum gradients but small coordinate gradient terms, would cause the optimization to focus on reducing the larger terms first to reduce the loss function value. This step causes the optimization to drift towards the wrong solution and get stuck in a local minima of the loss function. Problems like this are usually dealt with by normalizing each gradient element independently. However, normalizing Hamiltonian phase-space data like this will not guarantee that the volume is conserved. Despite these difficulties, we will attempt to construct a method similar to Champion et al. [6] that reaches sufficiently accurate results, replacing classical SINDy with Hamiltonian-SINDy.

While testing the algorithm, Zygote takes unfeasibly long to differentiate through the *RuntimeGeneratedFunction* for the SINDy vector field. Due to these inherent limitations in the *Zygote* package, we used *Enzyme* [32] for automatic differentiation in this part. Although Enzyme has specific requirements for storing gradients and certain incomplete features, its performance increases were enough to justify tackling these obstacles.

In the prototyping stage, we initialize the encoder-decoder pair weights to identity and start with the data already in Hamiltonian conjugate coordinates. This initialization allows

the same freedom for the encoder-decoder coordinates to fluctuate while increasing the likelihood that the correct coordinates are retained. For testing purposes, starting from a state close to the correct solution for the encoder-decoder weights helps us prove the concept of the method.

More straightforward loss functions were also tested. In one setting, the loss function was set up as follows:

$$Loss = (\sum_{i}^{N}(\varphi_D(\psi(\varphi_E(x(t_0)))) - x(t_1))^2)^{\frac{1}{2}} \tag{3.5}$$

Where φ_E is the encoder, ψ is the SINDy layer and φ_D is the decoder. In equation 3.5, the goal was for the encoder to transform the measured data at time t_0 into canonical coordinates, use Hamiltonian SINDy as in Section 3.2.1 to predict a state in the encoded space at a later time step t_1, and then transform that value to the original coordinate space with the decoder. This predicted value would then be compared against sampled data in the original coordinate space at time-step t_1. However, this performed worse than the loss function from Champion et al. [6]. Another approach could be to train the auto-encoder separately from the SINDy part. However, this does not guarantee that the encoded coordinates indeed constitute a pair of canonical-conjugate coordinates, even though the reconstruction error is small.

Despite these hindrances, there are a few possible ways to guide the encoder weights to return canonical-conjugate coordinates within limitations. A possible approach is to detect the symmetry inside systems. An example is a pendulum on a cart where the Hamiltonian is invariant to position. In the Lie algebra framework, this symmetry can be exploited to affect the weights of a neural network, as seen in the work from Dierkes et al. [10]. However, adapting, implementing and testing this approach is outside the scope of this thesis.

A last possible approach is to limit the number of basis functions the algorithm can choose from at the start. This last approach was also tested in our implementation and will be shown in the results. However, this relies on additional apriori knowledge of the system dynamics beforehand, which is only available in special circumstances.

This explanation concludes the discussion of the implementation of the code. It is finally time to discuss the results, where we can show the abilities and shortcomings of the methods discussed in this thesis.

4 Results

This chapter will discuss the results from each section of the Methodology. Examples will be shown from different areas of physics and how the developed SINDy extension methods perform on each. The various implemented methods will also be compared where necessary. The main goal is to show the power of Hamiltonian-SINDy while also showing an example to depict the approach's limitations. The results from Autoencoder-Hamiltonian-SINDy will be given through a test showing its current limitations and a proof of concept example.

We use random initial conditions integrated using reference model vector fields and the predicted SINDy vector fields for trajectory plot comparisons. The integrators we employ are either the *Tsit5* tableau of Runge-Kutta coefficients [43] or one of the Geometric Integrators available in the package *GeometricIntegrators.jl* [27]. Integrators that preserve geometric properties are popular when dealing with Hamiltonian systems, where, for correct predictions, certain properties, such as the symplectic structure, need to be conserved during a long-time simulation. From the *GeometricIntegrators.jl* package, we mainly use the *Gauss(2)* integrator, which employs the fully implicit Runge-Kutta method with Tableau Gauss [27].

Moreover, all noise added to plots is zero-mean Gaussian noise with various noise amplitude as described in Equation 3.1 to simulate real measurement data. In all the SINDy cycles run in the tests, the SINDy cycles never reach the maximum limit of 10, and the sparsification usually ends at a maximum of five loops. The sparsification threshold is usually set to 0.05 by default. If it is changed for any example, it will be mentioned. Finally, a single SINDy cycle indicates that only one loop of the SINDy sparsification and optimization was carried out after initial optimization. The full implementation of the code and all examples can be found in the Github repository here [1]

4.1 Classical SINDy

Starting with a simple test, we will compare the run-time of the original SINDy code from Brunton 2016 et al. [5] to our re-implementation of the SINDy algorithm in Julia. Both will use the least-squares algorithm on the Lorenz system example [30]. The original code is in Matlab Programming Language, while our rewritten and altered versions are in Julia. Julia usually has significantly shorter computation time for numerical optimization and root finding problems [9]. So Julia is expected to be faster for this example as well.

[1] GitHub Repository https://github.com/JuliaRCM/SparseIdentification.jl.

From the results of running a profiler for computation time and accuracy on the least-squares algorithm for the Lorenz system example five times, it was seen that, on average, the Matlab code takes approximately 0.211 seconds to run as compared to the Julia code, which runs in 0.021 seconds, ten times less. However, due to both the Matlab and Julia code being un-optimized for computation speed and the Julia code especially employing many data structures and supporting code to run other optimizers and methods, the run-times should not be held in high regard and are only mentioned in passing. The results are summarized in Table 4.1 where both implementations have similar accuracy for the predicted coefficients.

Language	Average Computation Time (sec)	Max Coefficients Residual	SINDy Loops
Julia	0.021	0.0004	3
Matlab	0.211	0.0001	10

Table 4.1: SINDy Metrics Julia vs Matlab for Lorenz System

Figure 1, in Brunton et al. [5], outlines how the data is obtained in the original version. In their code, state data is collected for a time series, and then noise is added while calculating the gradient of these states using the reference Lorenz system equations. We differ slightly in this approach by uniformly sampling from a range instead of using a time-series approach to get the states. Figure 1 part III from Brunton et al. [5] shows how the collected coefficients will be used along with the library of basis functions to identify a model, and we employ the same functioning.

To correctly judge the accuracy of the implementation in Julia versus the original Matlab code of Brunton et al. [5], the error versus noise plots for the Lorenz system will be compared to Figure 6 in Brunton 2016 et al. [5], ensuring the accuracy is of the same order or better. The actual and predicted dynamics trajectories are compared to get the error plots. The sum of the absolute errors between the reference and the predicted states are plotted in **log10** scale. The result of this endeavor can be seen in Figure 4.1.

Comparing our results to the ones from the original Matlab code, it can be seen that the errors are of the same order. The errors in Julia have been calculated with a step size of 0.001 but plotted with a step size of only 0.1 to allow the differences between noise inputs to be seen more clearly. It is assumed that a similar procedure was carried out for the original plot. The results show us that the re-implementation has very similar accuracy to the original implementation, while as was seen in Table 4.1, the Julia computation implementation in its current state is around ten times faster.

Next, we will plot dynamics from the linear-harmonic oscillator, Lorenz system, and mean-field model for fluid wake behind cylinder examples from the original paper [5] using the least squares and BFGS [3] optimizer. The first shows a simple case; the second shows a chaotic system, and the last shows that the SINDy method works on nonlinear partial differential equations. Plots for dynamics over time will be presented to see the drift in error. We will also discuss the dynamic equation coefficients to see if there are any extra or fewer basis functions and how much the terms differ from the reference solution.

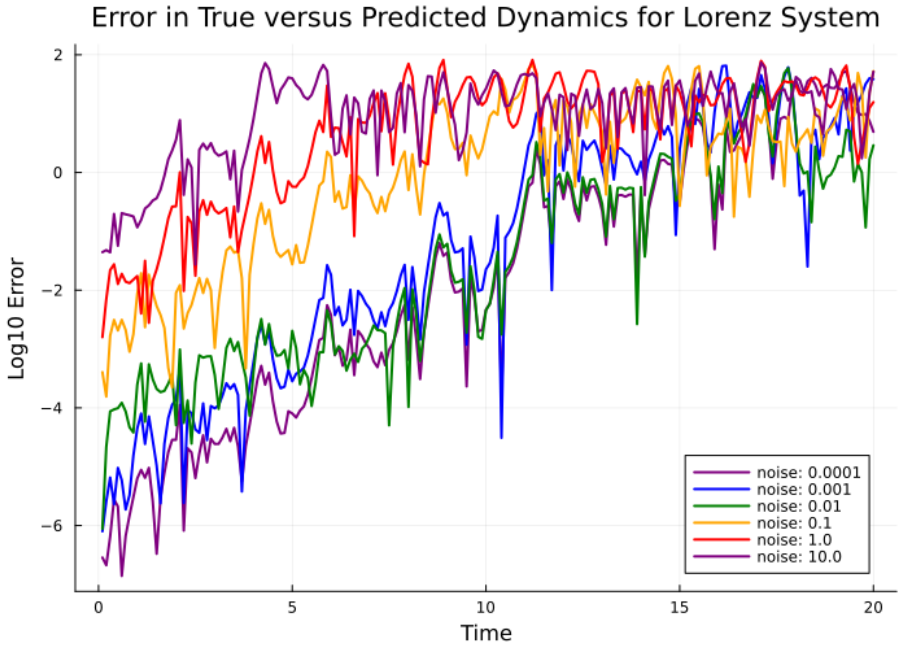


Figure 4.1: Error over time in states of the Lorenz system for various noise inputs

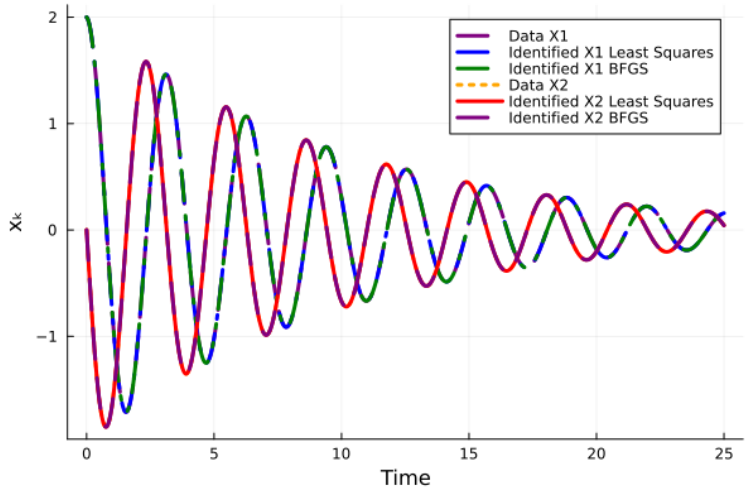
4.1.1 Illustrative Example: Two-Dimensional Damped Linear Oscillator

Starting with the linear oscillator example, we have the true equation as follows:

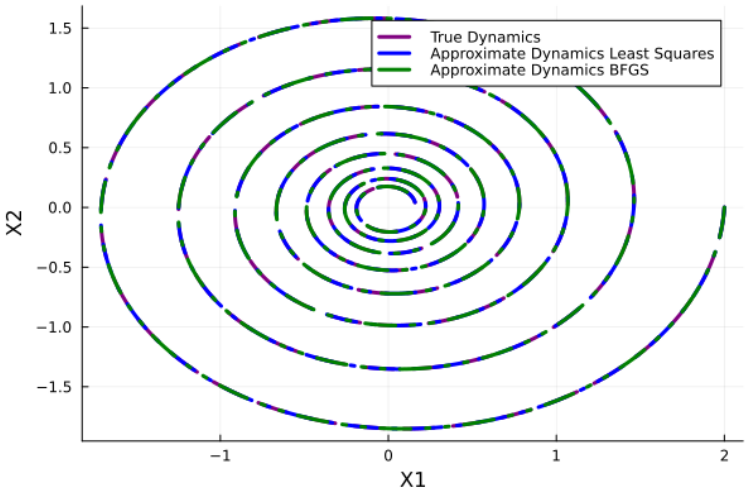
$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -0.1 & 2 \\ -2 & -0.1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} \tag{4.1}$$

On this model, SINDy was performed, with zero noise, using both the least-squares method and the BFGS optimizer. The results for dynamics over time and in phase space show that the predictions from both SINDy optimizations match the correct dynamics to extremely high precision.

When a noise amplitude equal to *1.0* is added, which is 5% of the sampling space that was *(-20, 20)*, one extra constant bias value from both methods is received. However, the coefficients for the correct basis functions are still accurate within one decimal place. The dynamics are reproduced to a reasonably accurate estimate, as was seen previously from the errors reported in Figure 3.2. The coefficients can be seen in the Appendix, Figure 1, which shows that the residual for the predicted and correct coefficients was small. The algorithm only takes two SINDy iterations each time for sparsification.



(a) Dynamics in Phase-Space



(b) Dynamics over Time

Figure 4.2: Linear Harmonic Oscillator

4.1.2 Chaotic System: Lorenz Attractor

Next, the Lorenz system's dynamics are depicted, showing the algorithm's performance on a chaotic system. The equation for this system are:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = x(\rho - z) - y$$
$$\dot{z} = xy - \beta z$$

Small differences in model coefficients or initial conditions can grow exponentially in the Lorenz system. However, capturing dynamics on the attractor is more important in such a system since it is chaotic, and any small variation causes exponential divergence [5]. As shown in the results in Figure 4.3 and the coefficients in the Appendix Figure 2, the dynamics are captured very precisely. For this example, the standard parameters are used, namely $\sigma = 10, \beta = 8/3, \rho = 28$, coupled with initial conditions [x y z]^T = [-8 7 27]^T for trajectory plotting. Uniform sampling is conducted from the area [-20, 20], and 3375 samples are collected. Then two noise variations are introduced to the system $\eta = 0.01$ and $\eta = 10$. Parameters and some metrics can be seen in Table 4.2.

Method	Noise	Max Coefficients Residual	Sparsification Threshold	SINDy cycles	Iterations
Least-Squares	0.01	1e-5	0.05	2	-
Least-Squares	10	1e-2	0.1	3	-
BFGS	0.01	1e-5	0.05	2	513
BFGS	10	0.003	0.1	3	288

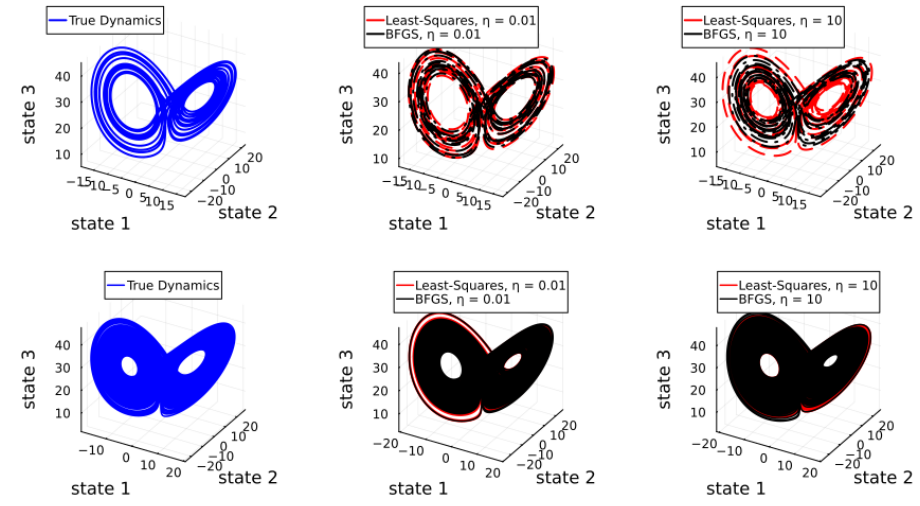
Table 4.2: Metrics and parameters for the Lorenz system

From table 4.2, we see from the approximate coefficients residual that the accuracy is approximately the same between the optimizers. The sparsification threshold was increased in the high noise regime to eliminate extra coefficients that were showing up. We see that the BFGS method takes more SINDy cycles, which increases with noise, indicating that the noise terms affect the sparsification, which is also seen from the need to change the threshold to remove extra coefficients. Finally, the total number of iterations is higher in the less noise regime as the loss function residual falls more there, reaching a value 3.437e-01. However, with high noise, the loss function residual can only fall to 3.429e+05 before stabilizing, which is much higher and reflected by the higher coefficients residual.

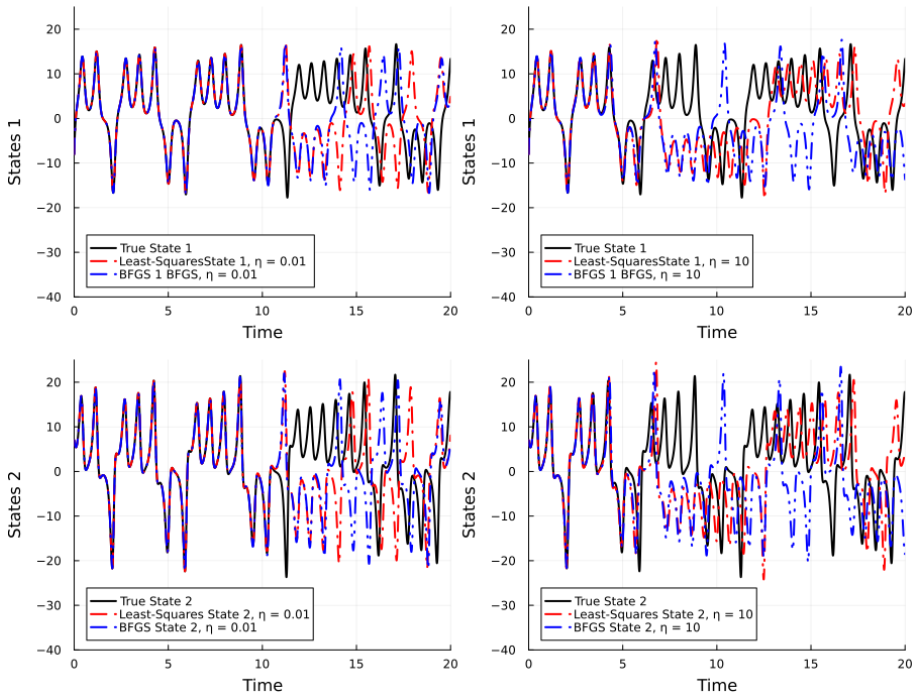
For trajectory plots, Figure 4.3 can be compared to Figure 4 and Figure 5 from Brunton et al. [5]. From an initial observation, both optimizers' predictions seem close to the reference trajectory. The predictions from BFGS and least-squares overlap well in low noise but differ more in the high noise regime, with least-squares performing slightly better, as is seen in the Dynamo view in Figure 4.3b. For long-term plots, we show that the shape of the trajectory is captured quite well by either optimizer, and the results look close to each other in Figure 4.3a.

4.1.3 Nonlinear PDE: Fluid wake behind a cylinder

Finally, the mean-field model dynamics [34] of fluid behind a cylinder are plotted. In this example, reference data was obtained from a fluid flow simulation past a cylinder



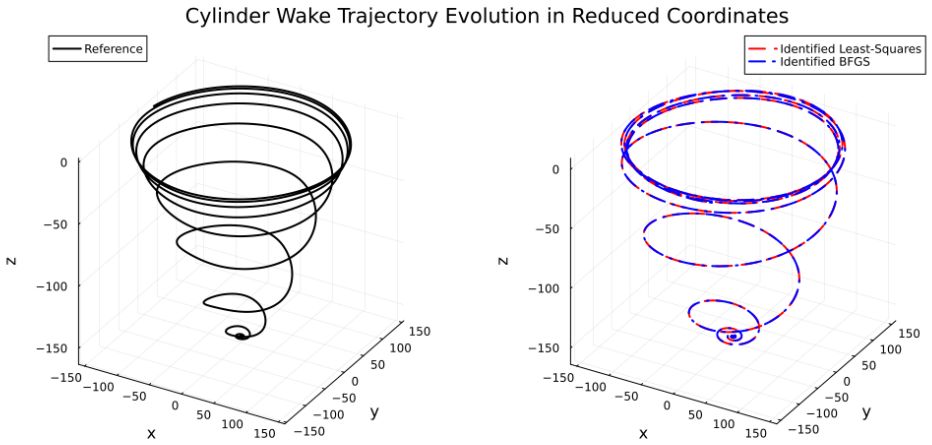
(a) Trajectories over time for t = 0 to t = 20 (top), and t=0 to t = 250 (bottom)



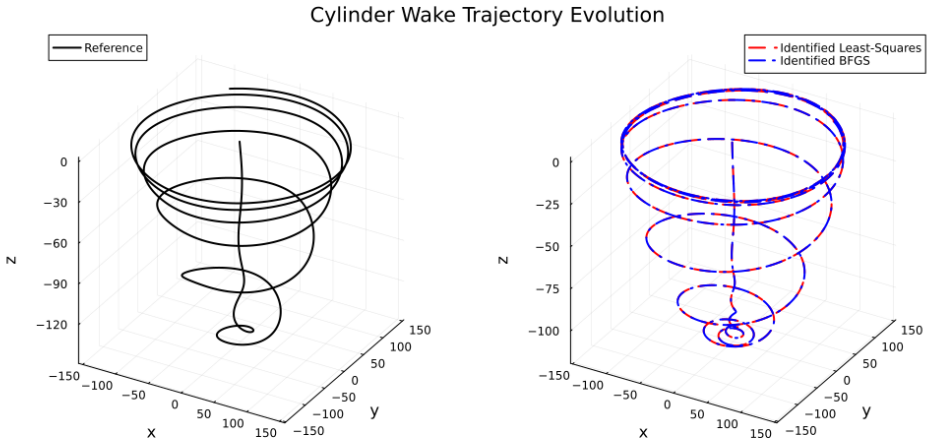
(b) Dynamo View of trajectories from t = 0 to t = 20

Figure 4.3: Lorenz System long-time trajectories and dynamo view for two noise levels

in 3 dimensions, at Reynolds number 100, using direct numerical simulation of the two-



(a) Evolution of wake trajectory in reduced coordinates



(b) Evolution of the wake trajectory initialized at the steady-state limit

Figure 4.4: Mean-field fluid-flow behind a cylinder for Reynolds number 100, in 3-dimensions

dimensional Navier-Stokes Equation. The three-dimensional system for a mean-field model described above has the following equations [5]:

$$\dot{x} = \mu x - \omega y + Axz$$
$$\dot{y} = \omega x + \mu y + Ayz$$
$$\dot{z} = -\lambda(z - x^2 - y^2)$$

As explained in Brunton et al. [5], the sparsification algorithms produce similar results to the data collected from simulations, and by relaxing the sparsity condition, almost perfect models can be obtained, although with higher-order nonlinear basis terms. For our case, the results obtained by the least-squares code are almost perfectly replicated using the *BFGS* optimizer in Figure 4.4 and their coefficient match almost exactly in Table 4. Only the no-noise case is simulated here as was done by Brunton et al. [5] as well.

Notably, the sparsification threshold was set to 0.00002, a minimal value, which was done to capture the higher-order dynamics of the model, allowing for some extra terms in the prediction. However, all the extra terms are only up to quadratic power, consistent with the Navier-Stokes equations [5]. Both optimizers take 4 SINDy loops. Unfortunately, as the coefficients' data for the reference model is not available, the residual coefficients cannot be directly compared. However, the BFGS optimizer reports a final loss residual error of 6.746e+04 from summing all the 5000 sample errors. However, comparing the plots in Figure 4.4, we can see that the prediction and reference results' trajectory plots match closely.

4.2 Hamiltonian SINDy

Four results will be discussed in this section. These cover a range of systems with dynamic equations covering polynomial, trigonometric, exponential, and logarithmic functions. Three of these systems are from different areas of physics, while one is a toy example for prototyping but still includes non-linearities. The first system models an oscillator of quadratic polynomial and trigonometric terms. The second example is a Toda lattice from solid-state physics, the third is a point-vortex system from fluid mechanics, and the last is a solar-system model. We will show different features of the method in each example, such as how changing noise and sparsity threshold affect the system, comparing Hamiltonian-SINDy with classic SINDy, and how extensive nonlinearities can affect trajectory predictions even with high accuracy coefficients. The last example will be to gauge the limits of the method, while the first three illustrate its range.

In most examples, we know the true Hamiltonian of the system. Still, we always start with a basis library of around 100 terms to show the system's identification ability. The goal is to find the correct dynamic terms, not just sparsification and system trajectory prediction. We will sample either from a uniform range or a normal distribution as convenient with the system under consideration. For comparison, one of the examples will be run with the gradient-less version of Hamiltonian SINDy from section 3.2.1 as well. Coefficients for all tests will be recorded in the Appendix. A long-time trajectory simulation will be carried out at the end of most examples to show how well the systems can function for predicting dynamics. To plot the long-time simulation, we will start with a random point from the data samples and integrate it up to a certain time.

4.2.1 Nonlinear Oscillator Model

This example Hamiltonian is made up for testing purposes and to present the method's power. It is a two-dimensional nonlinear oscillator system, i.e., four variables. Although its Hamiltonian looks simple, the system can have an intricate trajectory depending on the initial condition, as shown later. The Hamiltonian can be seen below in Equation 4.2:

$$H = \frac{1}{2}p_1^2 + \frac{1}{2}p_1^2 + cos(q_1) + cos(q_2) \tag{4.2}$$

We set up this system to search from 4^{th} order polynomial space, various trigonometric frequency terms, and functions for differences between variables, totaling 97 terms. This number and type of guesses for basis functions of the Hamiltonian are realistic as oscillatory trajectories with changing amplitudes are assumed to be prerecorded from measured data before setting up the library. We want sufficient basis library functions to meet the assumption that we started with little information about the system. We sample uniformly from the range [-20, 20] and take 10,000 samples with noise amplitudes of 2.5% and 10%. The BFGS algorithm will be used for optimization, with sparsification thresholds of 0.05 changed to 0.1 when the noise increases. The results obtained are as follows:

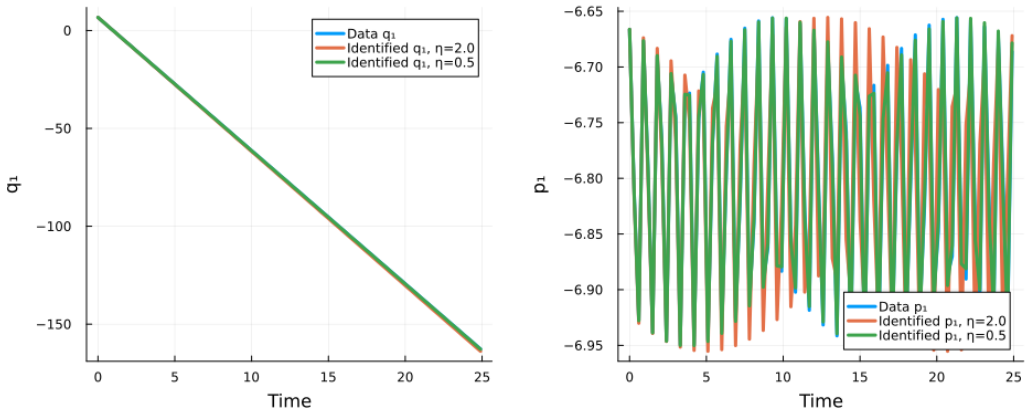
Noise	Sparsification Threshold	Max Coefficients Residual	SINDy cycles	Iterations
2.5%	0.05	0.01	1	115
2.5%	0.1	0.007	1	110
10%	0.05	0.04	1	118
10%	0.1	0.02	1	95

Table 4.3: Metrics and parameters for the nonlinear oscillator with BFGS

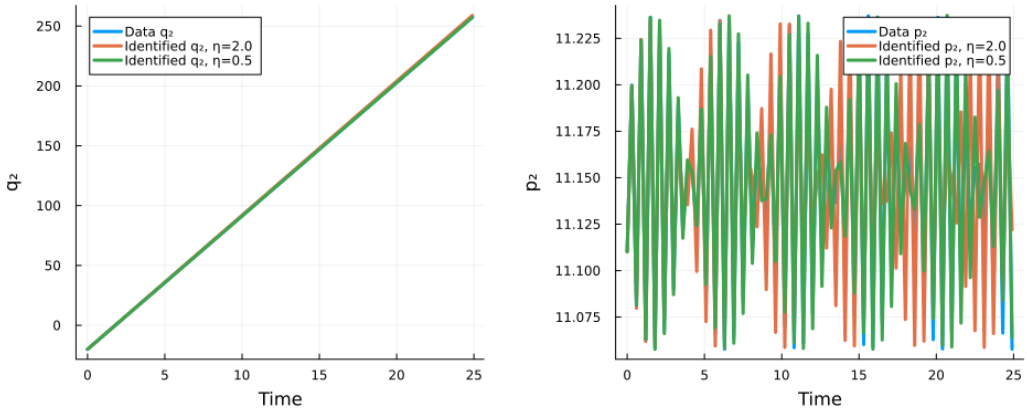
We get a sufficiently high accuracy for both the high and low noise inputs as seen from the coefficients residual reported in Table 4.3. Changing the sparsification parameter for the high noise input allows a better approximation of the actual dynamics, as noted by the decrease in the coefficients' residual. We note that the SINDy cycles are always equal to one, which could mean that the dynamics are easily identifiable as the system is a toy model. The total number of iterations carried out does not change much between the noise inputs, which indicates that the optimization, when coupled with the proper sparsification parameter, can filter through the noise easily. The coefficients' prediction only improved slightly from increasing the sparsification threshold. Still, it did cause the total iterations to decrease, meaning more extra coefficients were excluded before SINDy optimization. The accuracy of the low noise input also improves by increasing the sparsification parameter. From further testing with higher threshold values, the improvement in the prediction stops at a threshold of 0.2, which is about half the actual value of the smallest coefficient. This improvement precipice indicates that coefficients' weights are at first spread out and only experience significant shifting towards the correct prediction after sparsification.

We can see example trajectory plots in Figure 4.5, made with sparsification thresholds of 0.05 for low noise and 0.1 for high noise. As expected, the low-noise system overlaps

with the reference model trajectory more than the high-noise contaminated prediction. The difference in the predictions with high and low noise is mostly only in the amplitude of the trajectory and not the overall shape, which also indicates that the prediction errors are minor for both parameter settings. The predicted coefficients for these parameter sets can be seen in Figure 5 of the Appendix.



(a) Trajectories of position and momentum in the first dimension



(b) Trajectories of position and momentum in the second dimension

Figure 4.5: Trajectories of the nonlinear oscillator for noise inputs

4.2.2 Toda Lattice

A Toda lattice is a simple example of a many-body system because it contains exponential terms in its Hamiltonian, in which states interact. A Toda lattice is a model for a one-dimensional chain of solid particles [42] from solid-state physics. We will compare a

classical SINDy prediction with a Hamiltonian SINDy prediction here, along with some noisy inputs and threshold values results. We will use a four-particle system with a basis library containing 156 terms. We will then take 2000 data samples from a standard normal distribution. The equation for a Toda lattice Hamiltonian is:

$$H(p,q) = \sum_{n \epsilon Z} \left(\frac{p(n,t)^2}{2} + V(q(n+1,t) - q(n,t)) \right) \tag{4.3}$$

where $q(n,t)$ is the displacement of the n^{th} particle, $p(n,t)$ is its conjugate momentum, and $V(q)$ is the interaction Toda potential given by $V(q) = e^{-q} + q - 1$ [42]. We will start by comparing Hamiltonian SINDy to classical SINDy. We will use the BFGS optimizer and least-squares method to initialize the coefficients to zero for both methods, with a sparsification threshold of 0.05. All methods will have a noise input of at least 2.5% while the BFGS methods will have the same convergence criteria that the input coefficients' vector needs to converge to a value of $1e^{-8}$. The results can be seen for the states of one of the particles in Figure 4.6

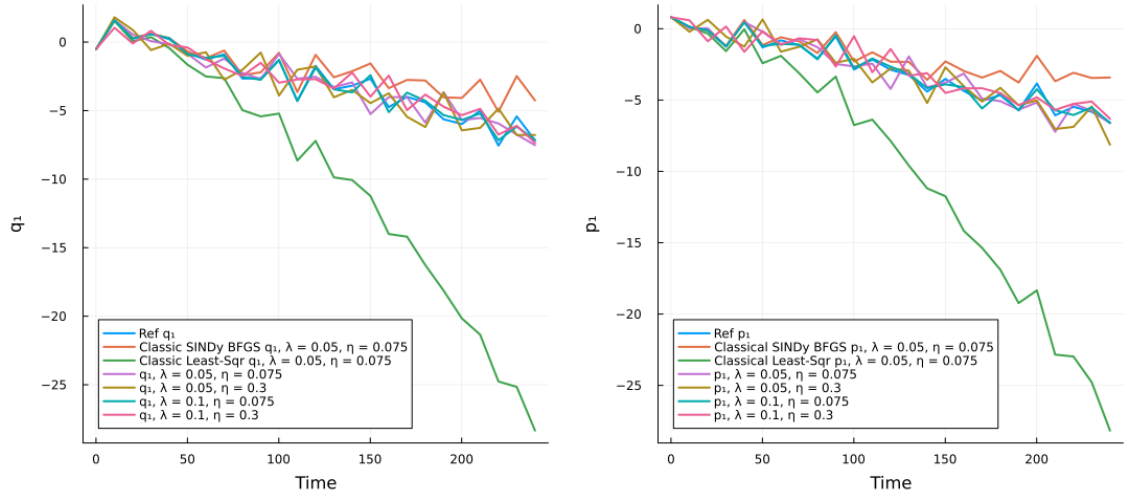


Figure 4.6: Trajectories of classical versus Hamiltonian SINDy for a Toda Lattice particle

We plot the trajectory using the implicit Runge-Kutta geometric integrator with Gauss tableau from the GeometricIntegrators.jl library with a time step 0.01 for 250 time units. As one can see, the Hamiltonian SINDy method produces much more accurate trajectory plots. Interestingly, the Newton-BFGS method took around 277 seconds for optimization, much more than the other methods, which consistently took 10 seconds. Below, the results are summarized in Table 4.4.

Both the classical SINDy approaches have the worst coefficient guesses for this case. This significant error is expected because they do not account for the symplectic structure

Method	Noise	Threshold Value	Max Coefficients Residual	SINDy cycles	Iterations
Classical-Least-Squares	2.5%	0.05	0.5	2	-
Classical-BFGS	2.5%	0.05	0.504	2	346
Ham-SINDy-BFGS	2.5%	0.05	0.006	1	140
Ham-SINDy-BFGS	10%	0.05	0.07	1	146
Ham-SINDy-BFGS	2.5%	0.1	0.003	1	140
Ham-SINDy-BFGS	10%	0.1	0.003	1	142

Table 4.4: Metrics and parameters for Toda Lattice Hamiltonian with BFGS

while predicting the coefficients. The high iteration number in classical SINDy BFGS and the much longer time it took to optimize show the difficulty of finding coefficients that minimize the problem without having the symplectic structure built into the predicted vector field. Figure 4.6 and Table 4.4 shows that the best results are obtained by pairing a higher threshold with lower noise. In comparison, the worst results come from using the least-squares method with a classical SINDy setup. All the Hamiltonian SINDy results are close together despite a case with 10% noise added to the data and a low threshold. The case with low noise and high threshold has the highest precision. The result indicates the advantage of placing the symplectic structure into the prediction method and choosing appropriate sparsification values.

4.2.3 Point Vortex

A point vortex is an entity where the vorticity is concentrated into a point. Roughly speaking, a vortex tries to make particles move along circular orbits. Vortices are found ubiquitously in nature, not just in fluids, which is why this is a useful example to test. A point vortex Hamiltonian consists of logarithmic terms, which is one of the reasons we choose it as an example. Its Hamiltonian is as follows [1]:

$$H = -\frac{1}{4\pi}\sum_{i=1}^{N}\sum_{j=1}^{N} p_i p_j log|q_i - q_j| \tag{4.4}$$

In this equation, p represents the strength of the vorticity, and q is the distance from its center [1]. We will change the Hamiltonian coefficient from $-1/(4\pi)$ to $1/2$ for easier analysis. We will use a 2-vortex system with 1000 data samples and 168 initial guesses for basis functions, sampling the data from a standard normal distribution. In this section, we will show the functioning of the Hamiltonian-nograd-SINDy method from section 3.2.1 to compare its performance. We will then plot trajectories for 250 time units with a time-step of 0.001. The results can be seen below in Table 4.5.

The classical SINDy methods perform the worst, even with zero noise, followed by the Hamiltonian SINDy without gradient approach from section 3.2.1 as expected. The classical SINDy BFGS method was set with the criteria to stop when the convergence in the coefficients inputs falls below $1e^{-8}$. The classical SINDy approaches are simply unusable for analysis because they cannot sparsify most of the coefficients from the basis library,

Method	Noise	Threshold Value	Max Coefficients Residual	SINDy cycles	Iterations
Classical-Least-Squares	0%	0.1	6.689	3	-
Classical-BFGS	0%	0.1	1.952	3	270
Ham-SINDy-noGrad	2.5%	0.1	0.268	1	400
Ham-SINDy	2.5%	0.05	0.0002	1	120
Ham-SINDy	2.5%	0.1	2e-5	1	93
Ham-SINDy	10%	0.05	0.003	1	96
Ham-SINDy	10%	0.1	0.002	1	134

Table 4.5: Metrics and parameters for Point Vortex

even when the prediction was made with zero noise input. This result again shows the importance of using the symplectic structure in the prediction. The without gradient Hamiltonian SINDy approach took an extremely long time for each iteration, so the maximum number of iterations was limited to 200 for each SINDy cycle. It also resulted in many extra coefficients. Due to the large errors in their predictions, the trajectories from these two methods could not be plotted. The trajectory plot from the case with 2.5% noise and 0.05 threshold also had to be excluded for reasons outlined below. Figure 4.7 shows the comparison plots

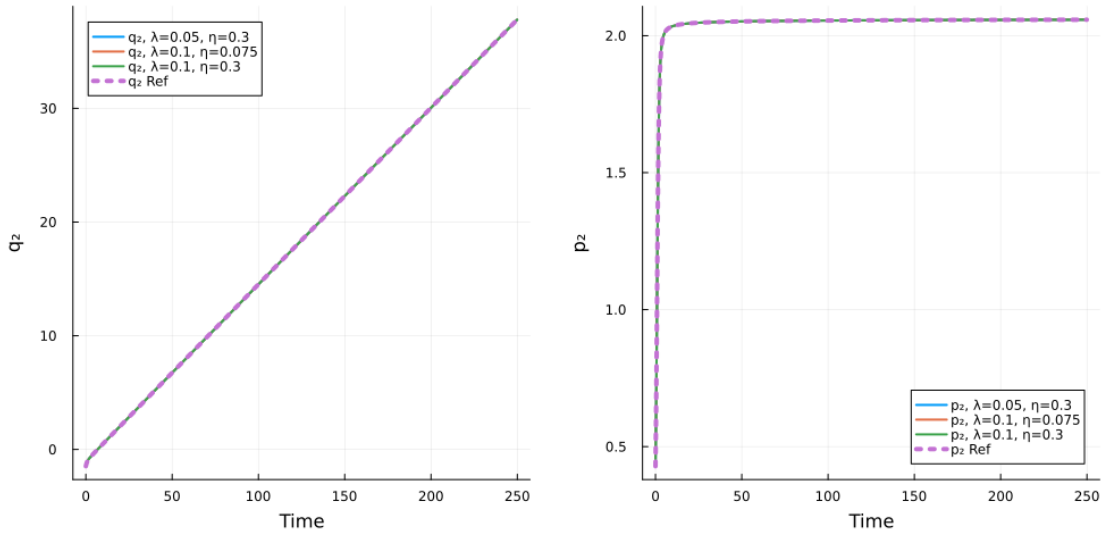


Figure 4.7: Trajectory plots of methods for a point Vortex

The reference trajectory plot matches the predictions quite well for the randomly chosen initial condition. From trying other initial conditions, it was also seen that prediction plots could also have a more significant difference from the reference. Despite also returning high precision predicted coefficient values, the case of low noise and small threshold could not be plotted in Figure 4.7 because of its significant trajectory error. Different parameter sets faced this problem during plotting for different initial state conditions. This

problem occurred due to the limitation of the integrator, which could not solve the trajectory correctly with the default constraints it was given. This initial condition and plot were chosen to highlight that the integrator setup matters as much as the correct prediction coefficients when plotting trajectories. The relevant coefficients for Hamiltonian-SINDy with noises and thresholds are reported in Appendix Figure 7, where it can be seen that all Hamiltonian SINDy predictions had high accuracy.

4.2.4 Solar System

The solar system model was initially chosen as an example to have a more tangible case for many-body interactions. However, it turned out to be an example where Hamiltonian-SINDy had problems functioning. We will write down the Hamiltonian for the n-body problem [12] and then use it to explain why it is challenging to solve using Hamiltonian-SINDy, for the case of celestial bodies.

$$H = \sum_{i}^{N} \frac{p_i^2}{2m_i} - \sum_{1<i<j<N} \frac{Gm_i m_j}{|q_j - q_i|} \tag{4.5}$$

Where m_i is the mass of each body being considered, usually on the order of the Earth's mass, i.e., 5.972 $\times$ 10^{24} kg and he gravitational constant is equal to G=6.6743 $\times$ 10^{-11} m^3 kg^{-1} s^{-2}. The problem is that for celestial bodies, the coefficient for the generalized momentum term $1/(2m_i)$ is minimal on the order of 10^{-24}. In contrast, the coefficient for the generalized coordinates term, $Gm_i m_j$ is huge, on the order of 10^{37}. Due to the sparsification algorithm, the cornerstone of SINDy, when we set up a library of basis functions, the sparsification threshold cannot retain just the correct function terms because of the huge difference in the coefficients and because of how small the coefficient $1/2m_i$ is by itself as well. Suppose one tries to make the Hamiltonian dimensionless. In that case, a huge coefficient of the product of the mass terms remains as a coefficient of the generalized coordinates, meaning sparsification is still a tough challenge. The difference between the correct coefficients is too large for the algorithm to output true dynamic equations.

We still try to run the algorithm for this case and examine the output, even if it is incorrect. For this example, we took data directly collected by the NASA Horizons System [28] for the position and velocity of celestial bodies at a specific time. The masses have been scaled in this data by 10^{24}. We will simplify the example to just an earth and sun system, with the sun's momentum and position held constant. Then, we integrate the system using the reference governing equations for two bodies to get more state data and reuse the reference equations on these states to get gradient data. Ultimately, we had 4000 data samples collected over a time period of $1e^6$ seconds by taking 250 time steps. Afterward, we set up a basis library of 113 basis functions. We try to keep the basis library size general but keep the types of basis functions in it close to the reference equation's basis functions. The results from the x-coordinate trajectory of a random initial condition for $1e^6$ seconds with a time step of 250 can be seen in Figure 4.8

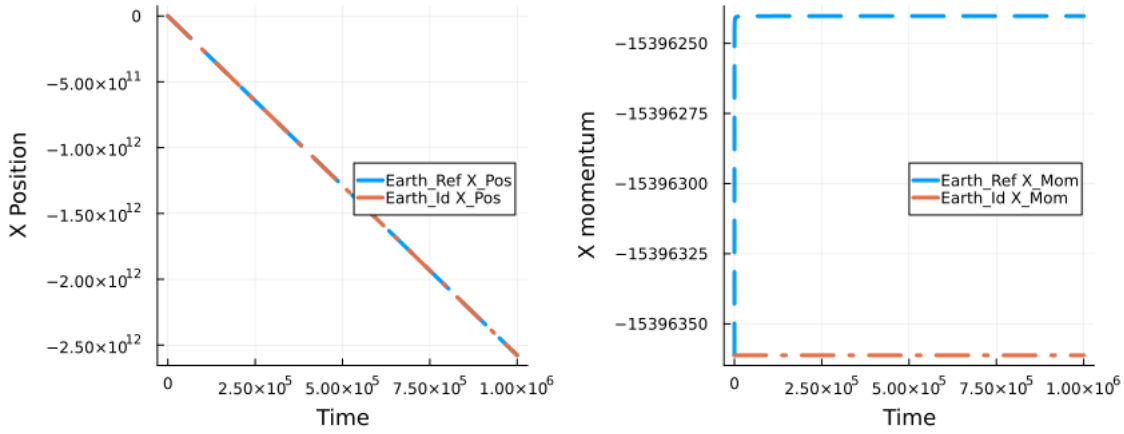


Figure 4.8: Earth Trajectory, identified (Id) versus reference (Ref)

It is immediately apparent from Figure 4.8 that the predictions for the Earth's momentum are incorrect. The coefficients found by the optimizer are correct for the momentum terms and zero for the position terms, as expected, due to the scaling of the masses. Since the momentum affects the coordinates gradient, the results were correct for position over time. However, since the momentum dynamics rely on the coordinates' terms for its gradient, the result is completely wrong for momentum over time, as expected.

This optimization takes around 3651 seconds to converge because of the prediction difficulty due to the differences in the coefficients' sizes. This example shows the limits of the method. However, obtaining better results in the Newtonian or Lagrangian formulations might be possible, where equations of motion look different.

4.3 Intrinsic Coordinate Identification

A version of the algorithm from Champion et al. [6] will be run on the damped linear oscillator and Lorenz system examples in this section. We will run tests by changing parameters like noise, sparsification threshold, batch size, and hyperparameter λ_3 explained in section 3.3. These parameters will then test how the prediction accuracy changes on the linear oscillator and Lorenz system examples. We will plot trajectories to compare errors, as getting the correct trajectory and not the exact dynamic equations is one of the original goals of this method. All automatic differentiation in this section uses the Zygote.jl package [21] with the Flux.jl package [22] for machine learning setups. We always use the Adam Optimizer [26], with Xavier initialization [14] and stochastic mini-batch technique to allow learning from diverse patterns. We will always divide the λ_1 hyperparameter by 10, one of the two values suggested by Champion et al. [6], the other being 100. This coefficient helps to scale the weighted effects of λ_1 and λ_2 hyperparameters against each other.

Finally, in both cases depicted here, the size of the intrinsic layer is kept the same as the input size for simplicity during development.

4.3.1 Damped Linear Oscillator with an Autoencoder

We start again with the damped linear oscillator example from section 4.1.1. The basis function library size is kept the same as before. The script will take 225 samples for the plot, sampled from the range (-20, 20). We run the optimization in three stages. The first is for 2000 epochs, then an optimization with SINDy cycles for each of 500 epochs, and a final optimization of 500 epochs. Table 4.6 shows the different parameter choices tested to find accurate trajectory predictions.

Number	Sparsification Threshold	Noise	λ_3 coefficient	batch size
1	0.05	0.5	0.52	32
2	0.05	2.0	0.52	32
3	0.1	0.5	0.52	32
4	0.05	0.5	0.22	32
5	0.05	0.5	0.82	32
6	0.05	0.5	0.52	64
7	0.05	0.5	0.52	128

Table 4.6: Test cases for Intrinsic Coordinate SINDy linear oscillator

In the figures below, these test cases will be referred to by their number in Table 4.6. We start by showing the trajectory results obtained from some of these cases. The trajectory is plotted for an initial condition of $x_0 = [2, 0]$ for 25 time units, with a time-step of 0.01. In Figure 4.9, we show plots for trajectories that were reasonably close to the reference. The trajectories that differed significantly from the reference are ignored. To make the plot, we pass the data through the encoder, integrate it using the predicted SINDy vector field, and pass it through the decoder. These steps are necessary because the SINDy vector field is for the encoded data, so the encoder-decoder layers must also be used while plotting the dynamics.

We can see from Figure 4.9 that test cases 1 and 5 produce high-accuracy results. For the corresponding parameters in Table 4.6, we note that choosing a small sparsification threshold, small noise input, and the smallest batch size while choosing a reasonably large λ_3 coefficient produces more accurate results. We also compare the loss graphs of all the cases in Figure 4.10.

We show three loss graphs from each part of the optimization. Test cases 1 and 5 consistently perform the best on close inspection of all three loss graphs. Initially, most test case losses fluctuate rapidly, except for test cases 1 and 5, which can be singled out as progressing steadily. Similarly, in the SINDy loss graphs, test cases 1 and 5 fall steadily across SINDy cycles, each of 500 iterations, with case 1 undergoing 3 cycles (1500 iterations), while most of the others undergo 2 SINDy cycles. From the final loss plot, we see that although all the losses have stabilized, the loss for cases 1 and 5 is much lower than the others, indicating a more accurate prediction.

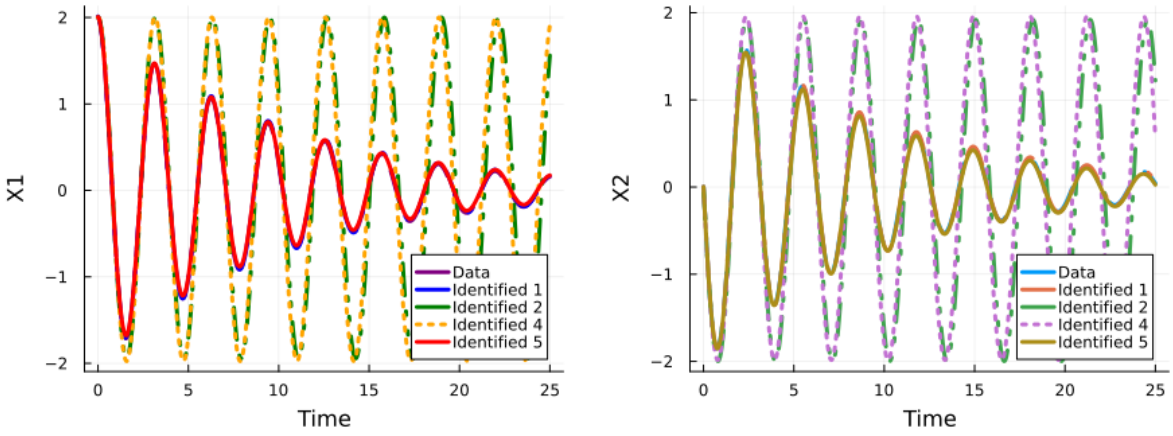


Figure 4.9: Trajectories for various test cases corresponding to Table 4.6

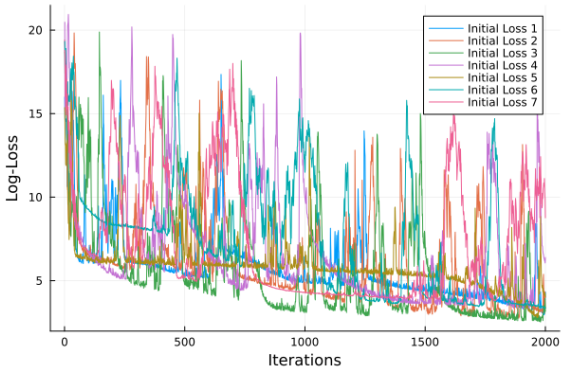
We show some prediction accuracy metrics in Table 4.7. We see that the iteration loss over all samples is a small value for these two test cases, which means the optimizer converged with high-accuracy parameters. The number of predicted coefficients corresponds to our choice for the λ_3 hyperparameter. In test case 5, a higher λ_3 value leads to fewer coefficients than the actual value of four, and test case 1 leads to 5 parameters. This difference hints that choosing a λ_3 value between these might improve the predicted results. The error in the trajectories is also tiny, as expected from the plots of these cases in Figure 4.9. The error is taken using the L^2 norm of the differences between the prediction and reference and dividing by the L^2 norm of the reference values. From this experiment, we conclude that the λ_3 parameter significantly affects the result. However, choosing the other parameters correctly also plays a role in correct prediction.

Number	Final Average Loss	Predicted coefficients	Trajectory Error
1	0.218	5	0.921
5	0.229	3	1.104

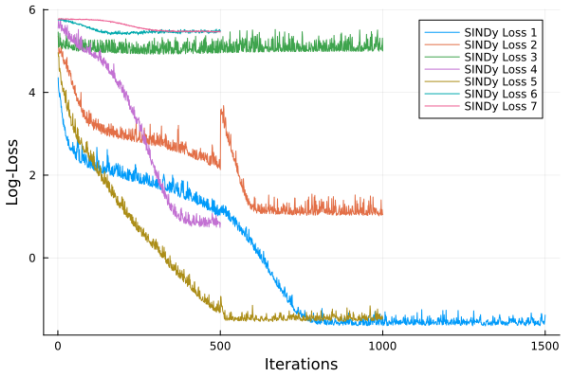
Table 4.7: Loss metrics of test cases for autoencoder linear oscillator

4.3.2 Lorenz system with an Autoencoder

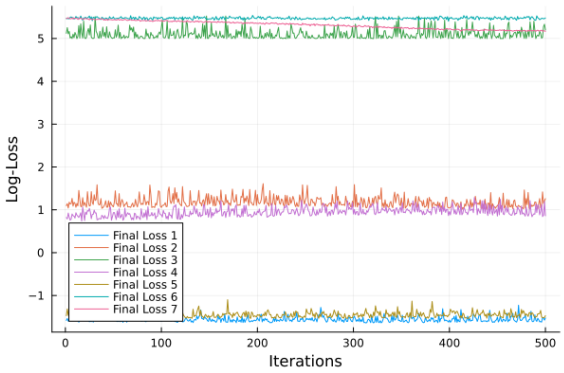
The Lorenz system will be used here mainly to test the accuracy of our setup because of the differences we introduced compared to the original setup from Champion et al. [6]. As mentioned in the methodology in section 3.3, we will use 3375 uniformly distributed samples. We introduce a minuscule noise input of 0.05% to the setup, starting with a library of 56 basis functions as before. Test cases are then set to find the best parameter combinations. Table 4.8 shows five parameter combinations that performed well. We will



(a) Initial losses



(b) SINDy losses



(c) Final losses

Figure 4.10: Loss graphs for Autoencoder Linear Oscillator test cases from Table 4.6

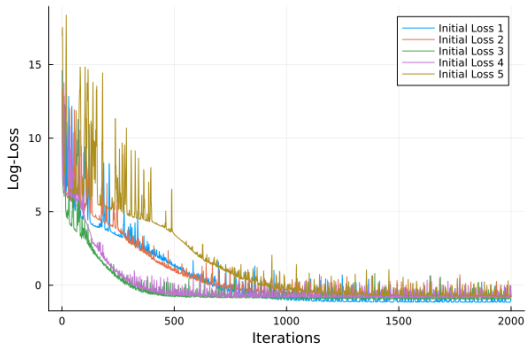
try to gauge the effects of these parameters on the prediction.

As shown in Figure 4.11 all initial loss function graphs stabilize well before we stop optimization. We made this decision because even when the losses stabilize, the values still

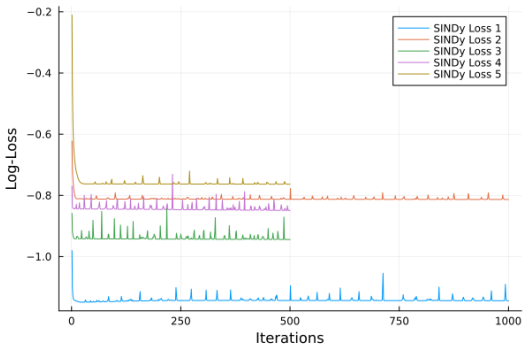
Number	Sparsification Threshold	Noise	λ_3 coefficient	batch size
1	0.05	0.01	0.62	80
2	0.05	0.01	0.92	80
3	0.05	0.01	0.92	32
4	0.1	0.01	0.92	32
5	0.1	0.01	0.92	80

Table 4.8: Test cases for Intrinsic Coordinate SINDy Lorenz System

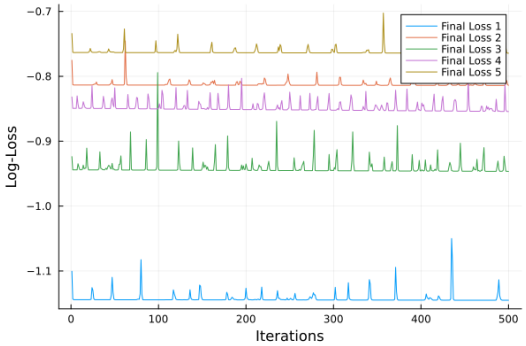
oscillate, and letting this continue for some iterations improves the result. This improvement could be due to the unnecessary coefficients falling below the sparsification threshold during these oscillations and being removed in each SINDy cycle later. We see that a combination of small threshold and large batch goes through the most SINDy cycles. These additional cycles can be attributed to fewer gradient updates with large batches, leading to less optimization and the smaller threshold eliminating fewer coefficients each time. The combination of a small λ_3 hyperparameter and large batch size from case 1 ends up with the lowest final loss, followed by case 3, which has a higher λ_3 hyperparameter and smaller batch size.



(a) Initial losses



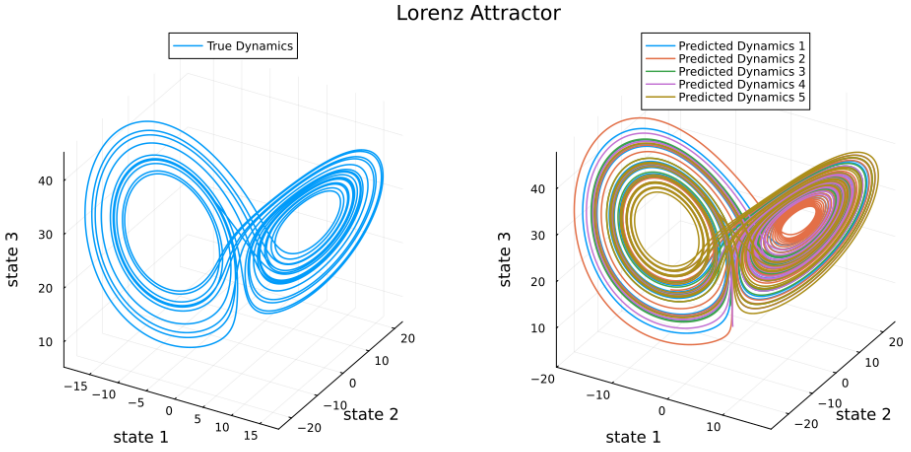
(b) SINDy losses



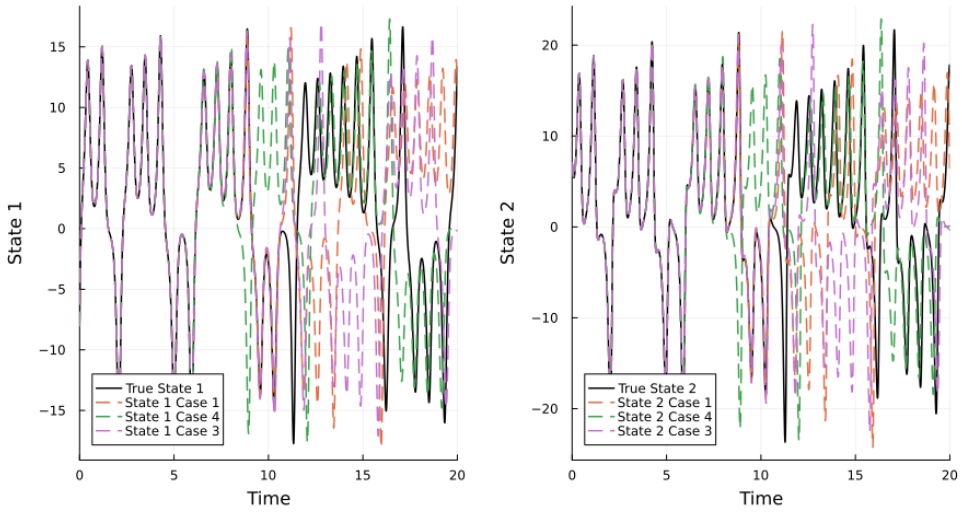
(c) Final losses

Figure 4.11: Loss graphs for Autoencoder Lorenz test cases from Table 4.8

Lastly, we see from Figure 4.12(a) that all the combinations perform similarly well. We pick three cases, case 1, case 4, and lastly case 3 which has the fewest coefficients, and plot them in Figure 4.12(b) to show that they all have high accuracy. The true Lorenz equation has seven coefficients. However, Case 1 and Case 4 have around 25 coefficients, which is likely how they can approximate the actual dynamics so well. Case 3 has the least with 16 coefficients, but we can see that it still captures the dynamics quite well.



(a) Trajectory plot wit all cases



(b) Dynamo trajectory plot with best cases

Figure 4.12: Lorenz System trajectory for case corresponding to Table 4.8

Compared to Champion et al. [6] who chose $\lambda_3 = 10^{-5}$, our λ_3 parameters are quite

high. This difference could mean that having an autoencoder that dramatically changes sizes in the hidden layers, as they had, coupled with an activation function, does not require a large sparsification hyperparameter. Similarly, their $\lambda_2 = 10^{-4}$ and $\lambda_1 = 0$ parameters were also chosen to be minuscule or zero, hinting that the autoencoder handles most of the job of correct coefficients discovery on its own without the SINDy doing much work. This effect could be due to the universal approximation power of neural networks when supplied large enough datasets, which is done in Champion et al. [6] as they use 256000 sample points, compared to our setup that uses only 3375 samples.

4.4 Autoencoder Hamiltonian SINDy

The Hamiltonian-SINDy algorithm will be combined with an autoencoder setup to discover canonical-conjugate coordinates and their correct dynamic equations. We will only show the nonlinear oscillator example in this section. The essential barrier in the optimization here is the conversion of the coordinates to canonical conjugate form. The constraints on the system need to be revised to allow correct dynamic equations to be discovered from a truly general library of basis functions. An initial test showing the problem of correct coordinate discovery will be developed as follows. We will set up the following basis array corresponding to the Hamiltonian in Equation 4.2.

$$\begin{bmatrix} cos(a_1 q_1 + a_2 q_2) \\ cos(a_3 q_1 + a_4 q_2) \\ 0.5((a_1 p_1 + a_2 p_2)^2 + (a_3 p_1 + a_4 p_2)^2) \end{bmatrix} \tag{4.6}$$

This basis array, when summed, is one of the closest basis library setups for the original Hamiltonian in Equation 4.2. The variable (a) comprises four coefficients, which can be optimized. To start the test, we will multiply the sampled data by an identity weight matrix **W** of size 2×2. This multiplication leaves the reference state and vector field data in their original canonical-conjugate coordinates form. We expect the coefficients in **(a)** matrix to be optimized to values corresponding to the inverse of **W**, generally in order to counteract the effect from the **W** matrix transformation. If we had instead placed the coefficients as just amplitudes of the library basis functions, the correct basis coefficients would not have been found, which was also tested. The effect of the **W** matrix on the data can be negated only by placing the coefficients in the arguments of the basis array. The amplitude coefficients are also fixed to the correct ones to make a more decisive point about how the optimization will function. Since **W** is initialized to identity which is its inverse, we expect the coefficients to converge to the following values in order to reproduce the correct Hamiltonian:

$$a = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = W^{-1} \tag{4.7}$$

We ran the optimization with 614656 samples from the range [-5, 5] and a sparsification threshold 0.05. We use the BFGS algorithm to compare the loss function's reference and predicted gradients. The coefficients matrix **(a)** is initialized by sampling random values from the standard normal distribution. We get the following results for the coefficient matrix after running the optimization:

$$a = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \tag{4.8}$$

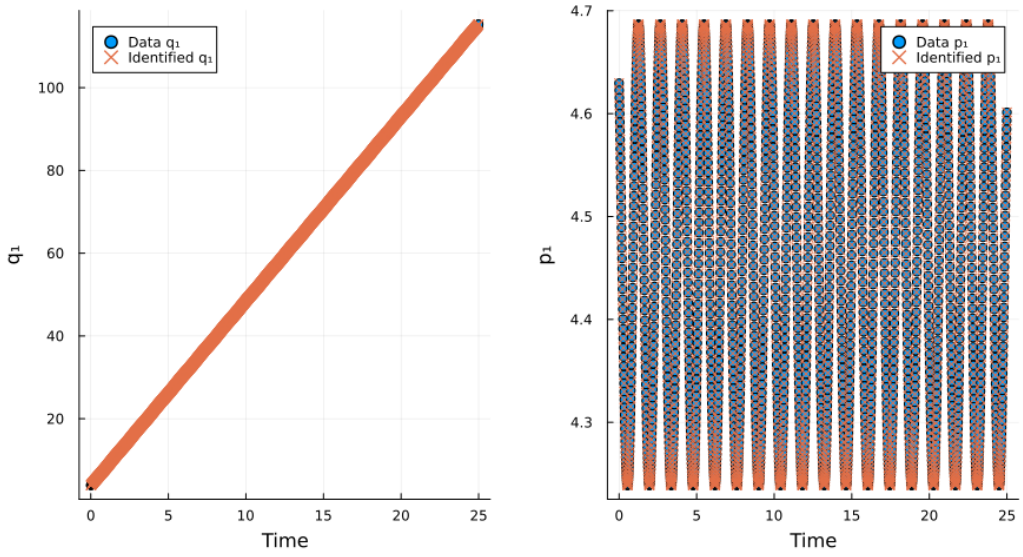
This coefficient matrix cannot negate the pseudo coordinate transformation from the **W** matrix because it is not the inverse of **W**. However, the coefficients still produce the correct vector field results, which can be seen by plugging in the values of **(a)** from Matrix 4.8 into the basis Array 4.6. As proof of its validity, it produces the correct trajectory for a sample initial condition, as seen in Figure 4.13

This test highlights the difficulty of finding the correct conjugate coordinates from transformed data, even when the exact basis is in the library with as few coefficients as possible and an enormous sample size.

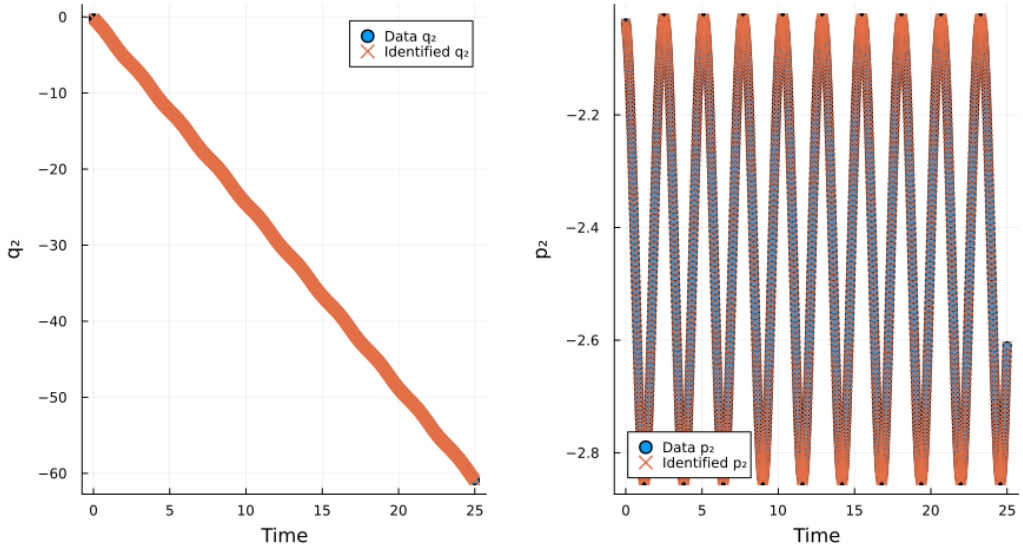
Therefore, we set up the nonlinear oscillator example with fewer functions in the basis library than before. We set up 22 basis library functions of up to 2^{nd} order polynomial terms and trigonometric terms for each variable. Each basis function is multiplied by a coefficient. We will also add coefficients to the arguments of each trigonometric function to allow further approximation capability, making a total of 30 coefficients to optimize. It was observed that taking data from the range [-20,20] is prone to causing crashes because of the gradient diverging to large numbers. Therefore, we will limit the uniform sample range to [-5, 5] and take 10,000 samples. We will not introduce noise into this system and keep the sparsification threshold of 0.05 and a batch size 256. We run the example for an initial optimization of 1000 epochs, then 500 epochs for each SINDy cycle, and the final optimization. The test will be run with coefficients initialized to ones as it results in better predictions. The sample data is already in canonical conjugate coordinates. The weight matrices for the encoder and decoder will be initialized to identity and the biases to zero in this example, which means they start with the values we want them to have. However, the algorithm can still adjust all encoder-decoder parameters, leaving freedom for generalization. We also tried random and Xavier initializations for the weight matrices, but the predicted results were consistently wrong by a significant margin with those initializations.

The result gives us encoder-decoder parameters extremely close to the values we want, as seen in Appendix Figure 8. The coefficients found are also approximately correct, with the highest coefficient difference being in the trigonometric amplitude terms. The predicted Hamiltonian can be seen below, which can be compared to the reference Equation 4.2

$$H = 0.519p_1^2 + 0.52p_1^2 + 0.7394cos(0.91834q_1) + 0.7396cos(0.91833q_2) \tag{4.9}$$



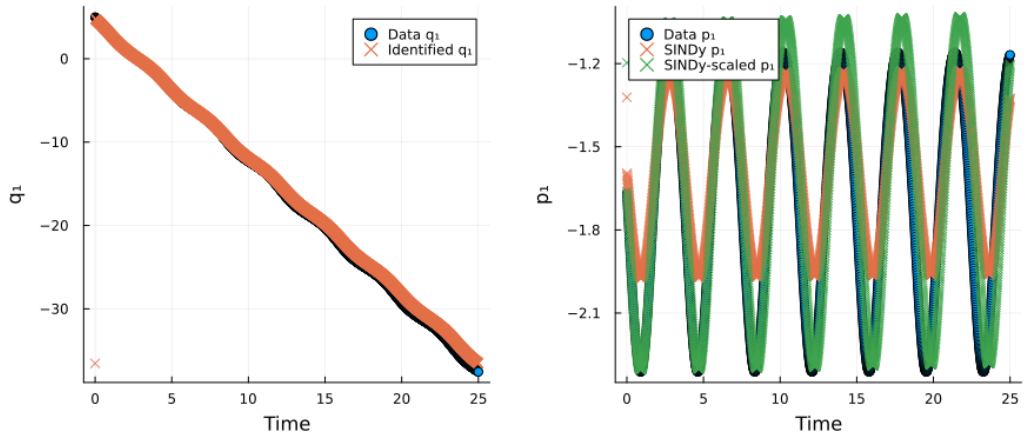
(a) State 1



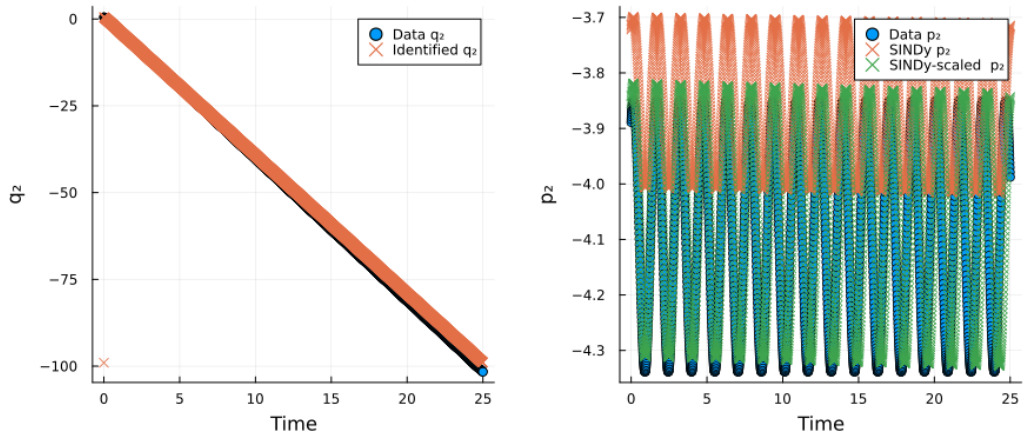
(b) State 2

Figure 4.13: Simple basis with encoder test on the nonlinear oscillator Hamiltonian

The resulting trajectories can be seen in Figure 4.14, where the prediction matches the reference reasonably well when the momentum predictions are scaled with small amplitude coefficients and bias terms.



(a) State 1



(b) State 2

Figure 4.14: AutoEnoder-Ham-SINDy trajectory plots for nonlinear oscillator Hamiltonian

In summary, these results tell us that it is generally challenging to get correct results from coupling an autoencoder with the Hamiltonian SINDy method due to its structure. However, the method can function reasonably accurately with a small basis library size corresponding to more system knowledge about the possible forms of the governing equations.

5 Conclusion

An extension of the SINDy algorithm for discovering structure-preserving Hamiltonian governing equations with limited conjugate-coordinate discovering capabilities was developed in this thesis. The Hamiltonian SINDy part of the method proved a viable option for small nonlinear Hamiltonian systems from various physics domains. At the same time, the intrinsic coordinate discovery section of the algorithm needs further development to be generally applicable. The results for long-term trajectory prediction of Hamiltonian systems clearly showed the advantages of having the symplectic structure built into the SINDy algorithm over the classical SINDy approach, which cannot predict correct long-term dynamics for Hamiltonian systems.

5.0.1 Outlook

This work can be extended in several directions. The most important would be to find a way to generalize the conjugate-coordinate discovery. Automatic symmetry detection through a Lie algebra framework [10] is a viable path for finding the intrinsic coordinates. After developing this method, tests could be run with the charge particle dynamics and similar example systems, where momentum is not just the product of mass and velocity. These tests will show if the algorithm can convert from velocity to conjugate momentum, finding the correct canonical conjugate coordinates.

Another possible extension could be to set up a similar SINDy method for Lagrangian systems to cover all three mechanics frameworks. This new setup would possibly open a direction for discovering governing equations of systems even in cases where library basis coefficients differ too much for SINDy to sparsify properly, by shifting to another framework where they might have a more suitable representation. This method extension could allow users to formulate the library functions more freely according to their known information, allowing easier use for discovering governing equations. It could also be possible that setting up intrinsic coordinate discovery and predicting trajectories for specific systems is easier in Lagrangian dynamics, in which case it would be helpful to develop a method for the Lagrangian framework.

A final extension could be to run this algorithm on large-scale problems while predicting truly long-term trajectories. Although we have shown here that the algorithm functions for various systems, it has yet to be proved that the method that we set up is functional when systems involving hundreds or thousands of bodies are to be solved over a vast time scale. A test of such a case would help fine-tune the method and prove that it is not limited to small and medium-sized systems.

Appendix

.1 Coefficients Tables

Row	1	2	3
1	1	0	0.14389222859109144
2	x	-0.09817519227406349	-1.9893327545336328
3	y	2.002389405402171	-0.09391527902705256
4	xx	0	0
5	xy	0	0

(a) Coefficients from Least-Squares

Row	1	2	3
1	1	0	0.16443976907027014
2	x	-0.0887889436251652	-1.989514926691221
3	y	2.0051700311240745	-0.09183980573224781
4	xx	0	0
5	xy	0	0

(b) Coefficients from BFGS

Figure 1: Basis coefficients of 2 Dimensional Damped Linear Oscillator with noise of amplitude 5% of max sample area

Row	1	2	3	4
1	1	0	0	0
2	x	-9.99997495633981	28.000008223879135	0
3	y	9.999993884698446	-1.0000179765270467	0
4	z	0	0	-2.666674683949839
5	xx	0	0	0
6	xy	0	0	1.0000010235867838
7	xz	0	-1.000001289162969	0
8	yy	0	0	0
9	yz	0	0	0
10	zz	0	0	0
11	xxx	0	0	0

(a) Coefficients from Least-Squares

Row	1	2	3	4
1	1	0	0	0
2	x	-10.000000060490533	27.999986520296765	0
3	y	10.000019154218759	-1.0000236090099348	0
4	z	0	0	-2.666666722174684
5	xx	0	0	0
6	xy	0	0	1.0000003897488272
7	xz	0	-0.9999993742010945	0
8	yy	0	0	0
9	yz	0	0	0
10	zz	0	0	0
11	xxx	0	0	0

(b) Coefficients from BFGS

Figure 2: Basis coefficients of Lorenz system for noise = 0.01

Row	1	2	3	4
1	1	0	0	0
2	x	-10.003321799548639	28.009523322507956	0
3	y	10.02958614022798	-0.9878614110366718	0
4	z	0	0	-2.6955658987296953
5	xx	0	0	0
6	xy	0	0	0.9996793752263947
7	xz	0	-1.0006240447405892	0
8	yy	0	0	0
9	yz	0	0	0
10	zz	0	0	0
11	xxx	0	0	0

(a) Coefficients from Least-Squares

Row	1	2	3	4
1	1	0	0	0
2	x	-9.986844414322725	28.000376107448613	0
3	y	9.990654419115735	-1.012121695980809	0
4	z	0	0	-2.6733143659882166
5	xx	0	0	0
6	xy	0	0	1.0001695770834804
7	xz	0	-0.9973008624695853	0
8	yy	0	0	0
9	yz	0	0	0
10	zz	0	0	0
11	xxx	0	0	0

(b) Coefficients from BFGS

Figure 3: Basis coefficients of Lorenz system for noise = 10

Row	1	2	3	4
1	1	-0.1159351711623463	-0.055618873777515516	-21.902217543433288
2	x	-0.009268691267008687	1.0346103832197067	-0.0008991437480045181
3	y	-1.0223337808595108	0.0046893454842392825	0.00005871198234094108
4	z	-0.0008729361550604438	-0.00043990653817875297	-0.3117685095461365
5	xx	0	0	0.0011382873621799388
6	xy	0	0	0.00021955236687555403
7	xz	0.0002112037508009073	0.00218683177673784	0
8	yy	0	0	0.0009142513518811318
9	yz	-0.0019300721457055148	-0.0017533369646682113	0
10	zz	0	0	-0.0010960213541481205
11	xxx	0	0	0

(a) Coefficients from Least-Squares

Row	1	2	3	4
1	1	-0.11593517113115923	-0.055618873792127946	-21.902217543492963
2	x	-0.009268691266988401	1.034610383219712	-0.0008991437480008133
3	y	-1.0223337808595179	0.004689345484232833	0.00005871198234381142
4	z	-0.0008729361548308538	-0.00043990653828750387	-0.31176850954697777
5	xx	0	0	0.0011382873621823297
6	xy	0	0	0.0002195523668759124
7	xz	0.0002112037508011444	0.0021868317767379177	0
8	yy	0	0	0.0009142513518836478
9	yz	-0.0019300721457056603	-0.0017533369646682933	0
10	zz	0	0	-0.0010960213541511242
11	xxx	0	0	0

(b) Coefficients from BFGS

Figure 4: Basis coefficients of Mean-field model of Flow Field Behind a Cylinder

Row	1	2
1	p[2]*q[2]	0.0
2	p[1]^2	0.5001698905321748
3	p[1]*p[2]	0.0
4	p[2]^2	0.4993910348322171
5	q[1]^3	0.0
6	sin(2p[1])	0.0
7	sin(2p[2])	0.0
8	cos(q[1])	1.0113993282078444
9	cos(q[2])	0.9929906392288812

(a) Noise level = 2.5%

Row	1	2
1	p[2]*q[2]	0.0
2	p[1]^2	0.4974293472687365
3	p[1]*p[2]	0.0
4	p[2]^2	0.4973826744858345
5	q[1]^3	0.0
6	sin(2p[1])	0.0
7	sin(2p[2])	0.0
8	cos(q[1])	0.9880782966024917
9	cos(q[2])	1.0069093945503544

(b) Noise level = 10%

Figure 5: Basis coefficients of Polynomial-Trigonometric Hamiltonian

Row	1	2	3	4	5
1	p[1]^2	0.510...	0.53...	0.49...	0.46...
2	p[2]^2	0.511...	0.53...	0.50...	0.47...
3	p[3]^2	0.492...	0.51...	0.50...	0.47...
4	p[4]^2	0.507...	0.50...	0.49...	0.48...
5	exp(q[1] - q[2])	1.001...	0.99...	1.00...	0.99...
6	exp(q[2] - q[3])	1.002...	0.99...	1.00...	1.00...
7	exp(q[3] - q[4])	0.999...	0.99...	0.99...	0.99...
8	exp(-q[1] + q[...	0.998...	0.99...	0.99...	1.00...

(a) Toda Lattice Coefficients with (noise=2.5%, threshold=0.05), (noise=2.5%, threshold=0.1), (noise=10%, threshold=0.05), (noise=10%, threshold=0.1) respectively

Row	1	2	3	4	5	6	7	8	9
1	p[1]	0.997...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	p[2]	0.0	0.994...	0.0	0.0	0.0	0.0	0.0	0.0
3	p[3]	0.0	0.0	0.99...	0.0	0.0	0.0	0.0	0.0
4	p[4]	0.0	0.0	0.0	0.998...	0.0	0.0	0.0	0.0
5	exp(q[1] - q[2])	0.0	0.0	0.0	0.0	-0.999...	0.99...	0.0	0.0
6	exp(q[2] - q[3])	0.0	0.0	0.0	0.0	0.0	-0.9...	0.999...	0.0
7	exp(q[3] - q[4])	0.0	0.0	0.0	0.0	0.0	0.0	-0.999...	0.999...
8	exp(-q[1] + q[4])	0.0	0.0	0.0	0.0	1.000...	0.0	0.0	-1.00...

(b) Toda lattice with classic SINDy least squares

Figure 6: Toda Lattice coefficients with different methods and parameters

Row	1	2	3	4	5
1	(p[1]^2)*log(abs(-p[1] + p[2]))	0.0	0.0	0.0	0.0
2	p[1]*p[2]*log(abs(q[1] - q[2]))	0.49996033...	0.49999...	0.49992...	0.499940...
3	p[1]*p[2]*log(abs(-p[1] + q[1]))	0.0	0.0	0.0	0.0
4	p[1]*p[2]*log(abs(-p[2] + q[1]))	0.0	0.0	0.0	0.0
5	p[1]*p[2]*log(abs(-p[1] + q[2]))	0.0	0.0	0.0	0.0
6	p[1]*p[2]*log(abs(-p[2] + q[2]))	0.0	0.0	0.0	0.0
7	p[1]*p[2]*log(abs(p[1] - p[2]))	0.0	0.0	0.0	0.0
8	p[1]*p[2]*log(abs(-q[1] + q[2]))	0.49996033...	0.49999...	0.49992...	0.499940...
9	p[1]*p[2]*log(abs(p[1] - q[1]))	0.0	0.0	0.0	0.0

Figure 7: Point Vortex Hamiltonian coefficients with (noise=2.5%, threshold=0.05), (noise=2.5%, threshold=0.1), (noise=10%, threshold=0.05), (noise=10%, threshold=0.1) respectively

Row	1	2	3	4
1	1.0571...	0.0000...	0.0001...	-0.000...
2	-0.000...	1.0570...	-0.000...	0.000...
3	-0.000...	-0.000...	1.0498...	0.000...
4	-0.000...	-0.000...	0.0001...	1.049...

(a) encoder weights

Row	1	2	3	4
1	-0.000...	0.0002...	0.0001...	-0.000...

(b) encoder bias

Row	1	2	3	4
1	0.906...	0.000...	-0.00...	0.000...
2	0.000...	0.906...	-0.00...	0.000...
3	0.000...	-0.00...	0.912...	-0.00...
4	-0.00...	0.000...	-0.00...	0.911...

(c) decoder weights

Row	1	2	3	4
1	-0.0001...	-0.000...	0.000...	-0.000...

(d) decoder bias

Figure 8: Autoencoder weights and biases on the nonlinear Oscillator with Hamiltonian SINDy

Bibliography

[1] Hassan Aref. Point vortex dynamics: A classical mathematics playground. *Journal of Mathematical Physics*, 48(6):065401, 06 2007. ISSN 0022-2488. doi: 10.1063/1.2425103. URL https://doi.org/10.1063/1.2425103.

[2] Vladimir Igorevich Arnol'd. *Mathematical Methods of Classical Mechanics*. Springer Science & Business Media, 2013. Vol. 60. Chapter 3.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[4] S. L. Brunton. Notes on koopman operator theory. *Universität von Washington, Department of Mechanical Engineering, Zugriff*, 30, 2019.

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

[6] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116 (45):22445–22451, 2019.

[7] Zhengdao Chen, Jianyu Zhang, Martin Arjovsky, and Léon Bottou. Symplectic recurrent neural networks. *arXiv preprint arXiv:1909.13334*, 2019.

[8] Hoang K. Chu and Mitsuhiro Hayashibe. Discovering interpretable dynamics by sparsity promotion on energy and the lagrangian. *IEEE Robotics and Automation Letters*, 5(2):2154–2160, 2020. doi: 10.1109/LRA.2020.2970626.

[9] Chase Coleman, Spencer Lyon, Lilia Maliar, and Serguei Maliar. Matlab, python, julia: What to choose in economics? *Computational Economics*, 58:1263–1288, 2021.

[10] Eva Dierkes, Christian Offen, Sina Ober-Blöbaum, and Kathrin Flaßkamp. Hamiltonian neural networks with automatic symmetry detection. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 33(6), 2023.

[11] Daniel DiPietro, Shiying Xiong, and Bo Zhu. Sparse symplectically integrated neural networks. *Advances in Neural Information Processing Systems*, 33:6074–6085, 2020.

[12] Robert W Easton. Introduction to hamiltonian dynamical systems and the n-body problem (kr meyer and gr hall). *SIAM Review*, 35(4):659–659, 1993.

[13] Reeves Fletcher and Colin M Reeves. Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154, 1964.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/glorot10a.html.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwozdz, Viral B Shah, Alan Edelman, and Christopher Rackauckas. High-performance symbolic-numerics via multiple dispatch. *arXiv preprint arXiv:2105.03949*, 2021.

[17] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. *Advances in neural information processing systems*, 32, 2019.

[18] E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the störmer–verlet method. *Acta numerica*, 12:399–450, 2003.

[19] Ernst Hairer, Ch Lubich, and G Wanner. Numerical geometric integration. *Unpublished Lecture Notes, March*, 1999.

[20] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric numerical integration*, volume 31 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 2006. ISBN 3-540-30663-3; 978-3-540-30663-4. Structure-preserving algorithms for ordinary differential equations.

[21] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018. URL http://arxiv.org/abs/1810.07951.

[22] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018. URL https://arxiv.org/abs/1811.01457.

[23] P. Jin, Z. Zhang, A. Zhu, Y. Tang, and G. E. Karniadakis. Sympnets: Intrinsic structure-preserving symplectic networks for identifying hamiltonian systems. *Neural Networks*, 132:166–179, 2020.

[24] Pengzhan Jin, Zhen Zhang, Aiqing Zhu, Yifa Tang, and George Em Karniadakis. Sympnets: Intrinsic structure-preserving symplectic networks for identifying hamiltonian systems. *Neural Networks*, 132:166–179, 2020.

[25] Anshul Joshi and Rahul Lakhanpal. *Learning Julia: Build high-performance applications for scientific computing*. Packt Publishing Ltd, 2017.

[26] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] Michael Kraus. Geometricintegrators.jl: Geometric numerical integration in julia. https://github.com/JuliaGNI/GeometricIntegrators.jl, 2020.

[28] Jet Propulsion Laboratory. Horizons system, 2023. URL http://ssd.jpl.nasa.gov/horizons.cgi. Accessed: 2023-11-26.

[29] B. Leimkuhler and S. Reich. *Simulating Hamiltonian Dynamics*. No. 14. Cambridge University Press, 2004.

[30] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of atmospheric sciences*, 20 (2):130–141, 1963.

[31] P Mogensen and A Riseth. Optim: A mathematical optimization package for julia. *Journal of Open Source Software*, 3(24), 2018.

[32] William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

[33] C. E. Mungan. Legendre transforms for dummies. *AAPT Summer Meeting, Minneapolis, MN*, 2014. URL https://www.aapt.org/docdirectory/meetingpresentations/sm14/mungan-poster.pdf. Poster presented at the AAPT Summer Meeting, Minneapolis, MN.

[34] Bernd R Noack, Konstantin Afanasiev, MAREK MORZYŃSKI, Gilead Tadmor, and Frank Thiele. A hierarchy of low-dimensional models for the transient and post-transient cylinder wake. *Journal of Fluid Mechanics*, 497:335–363, 2003.

[35] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[36] Jeffrey M Perkel et al. Julia: come for the syntax, stay for the speed. *Nature*, 572(7767): 141–142, 2019.

[37] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016. URL https://arxiv.org/abs/1607.07892.

[38] Henry J Ricardo. *A modern introduction to differential equations*. Academic Press, 2020.

[39] David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

[40] Benjamin Simons. Motion of a charged particle in a magnetic field, 2016. URL http://www.tcm.phy.cam.ac.uk/~bds10/aqp/handout_charged.pdf.

[41] John Robert Taylor and John R. Taylor. *Classical Mechanics*. University Science Books, Sausalito, CA, 2005. Vol. 1. Chapters 6, 7, and 13.

[42] Michael Eugene Taylor et al. *Mathematical Surveys and Monographs*. American Mathematical Society, 2000.

[43] Ch Tsitouras. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2):770–775, 2011.